Local Code Generation and Compaction in Optimizing Microcode Compilers

Steven R. Vegdahl

December 1982

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-82-153

Local Code Generation and Compaction in Optimizing Microcode Compilers

Steven R. Vegdahl

December 1982

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported in part by the Fannie and John Hertz Foundation and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-82-153	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) LOCAL CODE GENERATION AND COMPACTION IN OPTIMIZING MICROCODE COMPILERS		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) STEVEN R VEGDHAL		8. CONTRACT OR GRANT NUMBER(s) F33615-78-C-1551
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, Pennsylvania 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE December 1982 13. NUMBER OF PAGES 193
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research (NM) Bolling AFB, DC 20332		15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number).

Horizontal microarchitectures often have features that make it difficult for a compiler to produce good object code from a high-level language. Although the problem of compacting microcode into a near-minimal number of microinstructions has received a great deal of attention, other phases of the compiler have not been studied as thoroughly. This dissertation explores methods of *generating* quality microcode for horizontal microarchitectures, *compacting* the microcode, and the interaction between code generation and compaction.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block #20

There are often several code sequences that perform the same computation for a given microarchitecture. If the code generation and compaction phases of the compiler are executed sequentially, the code generator may not be able to determine the best code, because a code sequence that compacts well in one situation may contain several bottlenecks in another. This dissertation explores three methods of coupling the *code generation* and *compaction* phases of the compiler, and concludes that subtle micromachine features make it very difficult to produce good code unless the code generator actually produces several candidate code sequences that are compacted and compared with one another.

This dissertation also explores machine-independent methods of generating microcode. One aspect of the code generation problem—that of generating constants "intelligently"—is discussed in detail. A technique called *constant unfolding* is presented that can be used to produce code sequences that generate constants in "unusual" ways during execution; such code sequences often lead to more compact code when the literal field of the microinstruction is a bottleneck.

The classical microcode compaction problem is also examined. We show that this NP-hard problem can be solved in polynomial time if the number of registers in the micromachine is bounded, and use this result to argue that the problem is not general enough. A heuristic algorithm is presented for solving the general problem.

Abstract

Horizontal microarchitectures often have features that make it difficult for a compiler to produce good object code from a high-level language. Although the problem of compacting microcode into a near-minimal number of microinstructions has received a great deal of attention, other phases of the compiler have not been studied as thoroughly. This dissertation explores methods of *generating* quality microcode for horizontal microarchitectures, *compacting* the microcode, and the interaction between code generation and compaction.

There are often several code sequences that perform the same computation for a given microarchitecture. If the code generation and compaction phases of the compiler are executed sequentially, the code generator may not be able to determine the best code, because a code sequence that compacts well in one situation may contain several bottlenecks in another. This dissertation explores three methods of coupling the *code generation* and *compaction* phases of the compiler, and concludes that subtle micromachine features make it very difficult to produce good code unless the code generator actually produces several candidate code sequences that are compacted and compared with one another.

This dissertation also explores machine-independent methods of generating microcode. One aspect of the code generation problem—that of generating constants "intelligently"—is discussed in detail. A technique called *constant unfolding* is presented that can be used to produce code sequences that generate constants in "unusual" ways during execution; such code sequences often lead to more compact code when the literal field of the microinstruction is a bottleneck.

The classical microcode compaction problem is also examined. We show that this NP-hard problem can be solved in polynomial time if the number of registers in the micromachine is bounded, and use this result to argue that the problem is not general enough. A heuristic algorithm is presented for solving the general problem.

Acknowledgements

I wish to thank my advisor, Anita Jones, for much helpful guidance and encouragement during the course of this research. I would also like to thank the rest of my committee, Rick Cattell, Joe Newcomer and Guy Steele for their many helpful suggestions on improving this manuscript. I am especially grateful to Guy and his wife Barbara for their emotional support during times when the completion of this dissertation seemed an endless task.

I wish to acknowledge the following teachers, professors, and employers who played key roles in my scholastic development during my grade school, high school, or college years: Bill Wright, Maxeye Hanley, Francis Brewer, Ron Henley, Bill Colescott, and Don Knuth.

I am also grateful to the Fannie and John Hertz Foundation for its financial support during my years at CMU.

I wish to express my appreciation to the National Football League Players Association for going on strike during the final stages of the preparation of this manuscript, thereby permitting additional work on Sunday afternoons and Monday nights.

Finally, I wish to thank my wife Jeannie for much-needed love and emotional support.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

In 1951, Maurice Wilkes introduced the concept of microprogramming at the Manchester University Computer Inaugural Conference [Wilkes 51]. At that time, however, the cost of memory was sufficiently high that microprogramming was not used seriously in practice until more than a decade later with the implementation of the IBM 360 series machines [Fagg 64]. Since that time, the cost of memory, with its highly regular patterns, has decreased at a rapid rate, making it more attractive to implement digital systems in microcode. At the same time, programmers have demanded more complex computer architectures, which would be quite cumbersome to implement completely in hardware. Thus, microcode offers a number of advantages to both the hardware designer and the programmer:

Flexibility	Many decisions can be delayed much further in the design process.
Extensibility	Once an architecture is on the market, it can be extended with additional microcode, perhaps to tailor a machine to a special application.
Cost	The number of components (and types of components) can be reduced by implementing a digital system in microcode; the information density in the control memory is much higher than in combinatorial logic.
Simplicity	Many complex instructions, such as table translation and string comparison, are simpler to implement in microcode than in hardware.

The trend toward VLSI implementation of digital systems is expected to increase the use of microprogramming. The use of microcode rather than digital logic decreases hardware complexity, and increases functionality and flexibility. According to Parker and Wilner [Parker 81], "It is universally agreed that future single-chip processors will be microcoded."

1.1. Horizontal Microcode

The desire for high performance has led many micromachine designers to choose a *horizontal* instruction format [Husson 70, Salisbury 76], which is to say that for each machine resource there exists a field in the microinstruction that is wired to the control lines of the resource during the execution of that microinstruction. A vertical (i.e., traditional) machine instruction, on the other hand, specifies only a single operation to be performed. A vertical architecture may therefore be considered a degenerate case of a horizontal one.

Consider an example on a PDP-11. It takes three instructions to add two registers together, shift the result left one bit, and store the result in a third register:

```
MOV R2,R3
ADD R1,R3
ASL R3
```

In a horizontal architecture, it may be possible to compute the result in a single instruction because the shifter, the ALU function, and data paths are independently controlled. Figure 1-1 depicts a horizontal microinstruction format in which the shifter, ALU, and various registers are independently controlled, performing the above operation in single instruction.

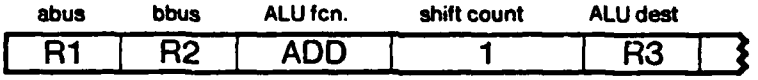

Figure 1-1: Horizontal microinstruction that performs an add and shift.

There may also be additional fields that allow the programmer to specify branching conditions or to control external devices.

Although the term *horizontal* technically refers to an instruction format with no encoding, a typical "horizontal" microinstruction format is a mixture of non-encoded and encoded fields. This often occurs because a particular resource or operation will be used so infrequently (in the designer's view) that the cost of an independent field is not justified. An example commonly found in microarchitectures is that of a branch address. It is not expected that a branch will occur during every instruction; similarly it is not expected that a every instruction will need literal (constant) data. In many microarchitectures, then, the *branch-address* field may specify literal data during microinstructions in which it is not specifying a branch address.

1.2. Motivation

Until recently, the production of microcode could be characterized by the following observations:

- The microcode was written by someone who was of necessity intimately familiar with the machine to be programmed—possibly the hardware designer.
- Once the microcode was written and tested, it was written onto a ROM, and not modified unless it was necessary to replace the ROM in order to remove a latent microcode bug.
- The size of the control store was relatively small, thereby bounding the complexity of the microprogram.

In the 1970's, however, it became increasingly popular to design machines that are programmed according to a different scenario [Nanodata 72, Fuller 76]. Microprogramming thus began to develop many of the same software engineering problems that traditional programming has had for the past two decades [Davidson 78]. In particular:

- A microprogrammer does not want to become familiar with the machine by studying circuit diagrams. It is desirable to free the programmer from having to learn the machine in extreme detail. At the very least, a tutorial describing the microarchitecture should be available. Ideally, the microprogrammer should be freed from understanding such details as propagation delays and data path routing.
- Microcode is frequently modified because many control stores are now writable. Thus, tools for reliably maintaining firmware are necessary. This can be especially important when a user desires to modify or extend "house-written" microcode, but keep it consistent with the rest of the system.
- As memory becomes less expensive, the size of control stores increases. Even "expert" microprogrammers are finding that the size and complexity of the microcode to be written and maintained is becoming too large [Jones 80].

In addition to the above problems which have analogues in macroprogramming, horizontal microprogramming also lends itself to pipelining. It is not uncommon to have parts of three or four unrelated computations being performed during a single microinstruction. For example, one microinstruction might contain a conditional branch on a comparison from the previous cycle, an addition being performed in the ALU, a main memory reference being initiated, and data from a register file being read onto a bus in preparation for being fed into the shifter on the next cycle. Such overlapping tends to make the code difficult to understand and maintain.

As user-microprogrammable machines become more common and control stores become larger, the effort required to produce and maintain microprogrammed systems increases. As a result, it is desirable to develop more powerful tools for the task. Researchers in firmware engineering have made progress in several areas.

Microprogram verification [Patterson 76, Carter 78], can be helpful in detecting inconsistencies that may be introduced during the production and maintenance of microprograms. Still, this approach does not free the programmer from writing microcode at the machine level.

The compilation of programs from a high-level language (HLL) has been quite successful in facilitating program development and maintenance in traditional software systems, so it seems reasonable to approach microcode in the same manner. HLL microprogramming does have drawbacks, however:

- Language requirements for microprogrammed machines may differ from those of traditional machines just as system implementation languages tend to differ from

application languages. For example, the pipelining that is possible in many microarchitectures can make it attractive to specify which branch of an *if-then-else* is most frequently executed [Fisher 81a]. DeWitt [DeWitt 76], Dasgupta [Dasgupta 78], and Patterson [Patterson 79] are among those who have explored solutions problems in the area of microprogramming languages.

- When a high-level language is used a compiler is necessary to translate the program into machine language. Because speed is often the motivation for putting a function into microcode in the first place, an optimizing compiler is desirable. There is still much work to be done in the area of horizontal microprogram optimization. This dissertation will explore several aspects of horizontal optimization.
- Validation of microprograms, which is sometimes done using oscilloscopes and logic analyzers, can be quite difficult. Code motion and other optimizations performed by a HLL compiler may compound this difficulty. There is certainly a need for microprogram validation/debugging tools.

Microcode compaction has been attempted with moderate success by a number of researchers [Yau 74, Tsuchiya 74, Dasgupta 76, DeWitt 76, Tokoro 78, Mallett 78, Wood 79a, Fisher 79, Ma 80, Landskov 80, Poe 80]. Compaction algorithms have typically assumed that the object code has been generated (either by a compiler or by hand), but has not been compacted. The goal, then, is to rearrange the given object code into as few instructions as possible without changing the semantics of the program. Although the problem is NP-hard (as will be shown in Chapter 2), a number of linear or near-linear algorithms have been devised that produce less than optimal results, but nevertheless appear to compact microcode quite well. Unfortunately, these algorithms exhibit a dependence on the initial ordering of the source code, as will be shown in Chapter 7.

1.3. This Research Effort

While much work has been done in the area of compacting already-generated microcode, relatively little attention has been paid to the problem of *generating* high quality microcode. Previous work assumed that the code had already been generated—either by hand, or by a previous phase of the compiler. In cases where a code generator actually exists (and the details of the generator are given), there is little evidence that an attempt was made to produce good code—the authors were concentrating on the compaction problem [Mallett 78, Fisher 79, Poe 80]. This dissertation concerns itself with certain aspects of the code generation process itself—in particular, generating code that is conducive to being compacted well.

Because is it generally agreed that the compilation process is too complex to perform in a single step [Aho 77, Leverett 79], we are presuming a compiler that consists of a number of steps, or *phases*. The premise on which this thesis is based is that the code generation and

compaction phases of the compiler cannot be separated if good code is desired; the two phases must be performed together, iteratively or in some other manner that allows the code generator some knowledge of how the code is being compacted. We have built code generation and compaction phases as part of this research effort, and have demonstrated that their coupling can improve code quality.

Other issues relating the generation of "packable" microcode are also discussed, but only to the degree that they relate to the primary topic. The intelligent generation of literals in microarchitectures has some potential benefits and is discussed in moderate detail.

The techniques described in this dissertation have been implemented in Pascal and have run on a DEC VAX-11/780 [Strecker 78]. Appendices A and B are devoted to the details of the implementation, and may be skipped by the casual reader.

1.4. Organization of the Dissertation

The first four chapters are of an introductory nature. Chapter 2 is an overview of the key issues in microcode optimization as we see them. The chapter is more or less a reply to the question: *Why is microcode optimization different from traditional optimization?* Chapter 3 is a review of previous work done in the field of microcode optimization; it describes the current state of the art in terms of the issues discussed in Chapter 2 and sketches the recent work by several researchers in the field. Chapter 4 describes in detail the issues addressed in this dissertation. In addition, it describes important related problems not addressed, along with the reasons for not addressing them. The chapter concludes with an brief description of the three techniques for coupling code generation and compaction that are considered in this dissertation.

Chapters 5 through 8 describe the work we have performed. Chapter 5 is a discussion of the micromachine model used in the implementation. It includes a discussion of the important features of microarchitectures that the model fits, as well as examples of micromachines which do not fit the model and the reasons for excluding them. It concludes with a discussion of the ramifications of the model for some of the issues stated in Chapter 2. Chapter 6 describes the heuristic search algorithm used in implementing the code generator. The chapter first describes the algorithm nondeterministically and then discusses the pruning mechanisms used that enabled it to run on a deterministic machine. In Chapter 7 we show that the commonly accepted compaction model is insufficient in at least two respects, and then present our algorithm, which solves a more general problem. Chapter 8 describes the three methods used to couple the code generation and compaction phases of the compiler and presents the experimental results for each method. The chapter concludes with a description of an attempt to combine the techniques.

Finally, Chapter 9 evaluates the research and summarizes what we believe to be its major contributions. Recommendations are also made for promising avenues of future research.

Chapter 2
Issues in Microcode Optimization

Over the past two decades, compiler writers have developed code optimization techniques that have been used in the production of a number of high-quality compilers [Lowry 69, Wulf 75, Kernighan 78]. In considering the problem of producing high-quality microcode, it is natural to try using the body of optimization knowledge that exists for traditional compilers. A number of microcode *compaction* systems assume that there exists an optimizing compiler that produces object code suitable for input to the compaction phase [Tokoro 78, Fisher 79, Poe 80].

If microarchitectures were sufficiently similar to traditional architectures, the idea of using a traditional optimizing compiler before doing the compaction would be a good one. Unfortunately, such architectures have characteristics in which many traditional optimization techniques either are ineffective, or require modification.

The scope of this chapter is much broader than that of the dissertation, including such issues as flow analysis, register allocation and short-circuit evaluation. We begin by discussing the key differences between microcode and traditional architectures. Following this, a number of traditional optimization techniques are evaluated with respect to their suitability for use in an optimizing microcode compiler.

This chapter has two purposes. The first is to acquaint readers, familiar with traditional architectures, with some of the optimization issues that arise when a horizontal architecture is considered, in order to give them a foundation from which Chapters 3 and 4 can be understood. The second is to bring the issues to the attention of researchers in the area of microcode optimization, many of whom have thus far concentrated on the issue of microcode compaction.

2.1. Differences between Microcode and Traditional Architectures

Most compiler optimization research has assumed a target architecture that is both *macro*—the instructions are stored in the main memory of the machine—and *vertical*—each instruction performs a single operation. The architectures that we are considering, on the

other hand, are *micro*—the instructions are kept in a high-speed local memory—and *horizontal*. We intend to describe how each of these aspects affects compiler optimization. The discussions in this chapter apply to *vertical microarchitectures* [Digital 78] and *horizontal macroarchitectures* [FPS 82] to a lesser extent.

Our research has led us to conclude that there are four major differences between horizontal microarchitectures and vertical macroarchitectures. First, the instruction format of a horizontal architecture allows independent computations to be performed during the same instruction. Next, the cost of a main memory access is more expensive on a micromachine, relative to the cost of instruction execution. Third, microarchitectures often require the programmer or compiler to be concerned with low-level timing details. Last, horizontal microarchitectures tend to have a large number of heterogeneous registers.

2.1.1. Horizontal instruction format

Traditonal machine architectures have what is known in the microprogramming literature as a *vertical* instruction format, while microprogrammable architectures that we are considering have a *horizontal* instruction format. The term *horizontal* came to be used because the instruction in such a machine has a large number of bits; that is to say, the instruction is typically a very wide (or horizontal!) one (see Figure 2-1).

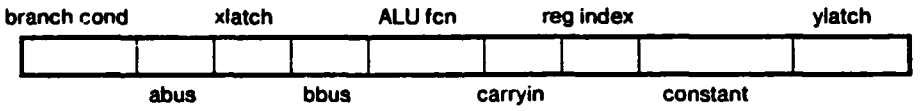

Figure 2-1: Typical horizontal instruction format.

The term *vertical* was then used to describe instruction formats that are not *horizontal*—the traditional instruction format in which there is an opcode and (possibly) one or more operands.

In a purely horizontal, or non-encoded, instruction format, the microinstruction (μI) is divided into a number of independent *bit fields*, where each field directly controls a machine resource. For example, the μI in Figure 2-2 contains seven fields. The first controls the ALU function; the second is the "count" input to the shift unit; the third and fourth serve as selectors for the ALU data input; the fifth selects a register from the register file reading or writing; the sixth specifies whether the register file is to be written; the seventh selects a condition for micro-branching. During the execution of every μI, each resource is controlled separately.

In microprogramming literature, the contents of an individual field of the μI is called a

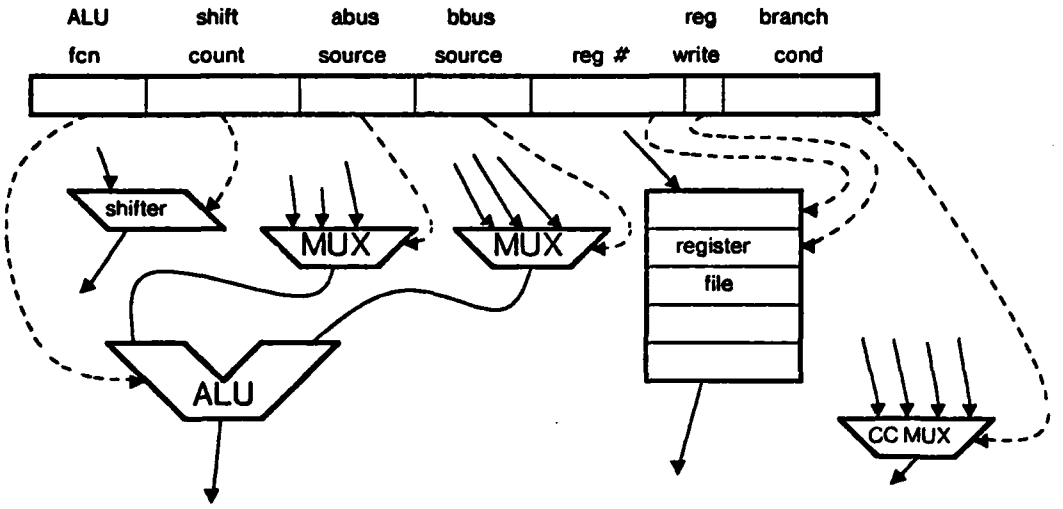

Figure 2-2: Horizontal control word controlling typical hardware resources.

microoperation (μOp). Because each μOp is an independent field, the μOp is *logically* the atomic unit of execution. A microcode generator, then, produces *μOps*, which are then compacted into *μIs*, which are *physically* the atomic unit of execution. Vertical architectures do not have a distinction between *logical* and *physical* atomic execution units.[1]

This distinction makes it necessary to compact the μOps into μIs after they have been produced. It is, of course, not generally possible to place all μOps into the same μI, because two μOps may require the use of a common hardware resource or μI field; data dependencies may also dictate that one μOp precede another. The compaction problem has been given a great deal of attention by researchers, and near-linear time algorithms have been discovered that usually do a good job compacting a μOp sequence into μIs for some microarchitecture models.

A horizontal instruction format also makes it difficult to predict the cost of a μOp. When compiling for a vertical target architecture, it is relatively easy to estimate the cost of adding a particular instruction to an existing code segment. The insertion of an `ADD #3, R0` PDP-11 instruction into a segment of code increases its execution time by a fixed amount and increases program size by two words. This cost of adding a μOp to a segment of code on a horizontal machine is not as easy to predict because other μOps may or may not be able to execute in parallel. The incremental cost of a μOp may be zero—if it can fit into an otherwise

[1] Several traditional machines do have instructions that are in some ways horizo- . T. PDP-8 and HP-2100 each have a special instruction in which several independent actions may be applied to the cumulator. PDP-11 instructions that use the *auto-increment* addressing mode may be considered "horizontal" in the sense that the auto-increment can be either invoked or not when a register indirection is occurring.

unused μI field—or large—if it requires one or more μIs to be added. It is generally not possible to determine such costs until the code has been compacted.

Thus a compiler for a vertical architecture can generally assume that the *cost* of an instruction is independent of the instructions surrounding it in the program. With these estimates, it is able to make intelligent decisions about whether or not to perform a code motion, how to allocate registers, and so forth. Such estimates are more difficult to make when compiling for a horizontal architecture because the cost is less predictable.

2.1.2. Cost of main memory access

The cost of accessing main memory is generally much greater on a micromachine than the cost of instruction execution. Macromachines tend to make one or more main memory references per instruction just to read the instruction itself; micromachines, on the other hand, typically fetch instructions from a high-speed internal memory. This difference is likely to affect compiler optimization strategies, such as register allocation, that might assume a main memory reference is relatively inexpensive.

2.1.3. Timing issues

The programmer of (or compiler for) a microarchitecture produces code that interacts more closely with the hardware than does code for a macroarchitecture. In particular, there are often timing constraints that require a programmer to be very careful in placing μOps into μIs.

Many microarchitectures have *polyphase* execution—in other words, different μOps can be executed during different clock phases within the μI; thus the execution of two μOps in a particular μI may or may not overlap. In addition, certain μOps may take longer than one μI cycle to execute, resulting in a situation where one μI begins execution before all μOps in the previous μI have completed. Finally, *volatile* registers—those registers that lose their values after a short period of time, such as one microcycle—are also common in micromachines.

2.1.4. Large number of storage classes

A typical macroarchitecture has a main memory, registers—some perhaps with special designations such as "stack pointer", "index register" or "program counter"—and possibly a processor status word and condition-code bits, with data being stored only in memory and registers. Microarchitectures, on the other hand, tend to have latches and registers of various lengths scattered across the machine. The Cm* Kmap [Ousterhout 78] has three 16-bit latches, one 12-bit latch, one 7-bit latch, two 4-bit latches and three 16-bit register banks along its various data paths; in addition it has several registers that contain data sent to/from main memory and external devices. The Puma [Grishman 78] has two 20-bit register banks,

two 60-bit register banks, four 60-bit latches, one 20-bit latch and three 12-bit latches in addition to registers for external communication. Research in compiler optimization suggests that a large number of register classes tends to make register allocation more difficult [Kim 79, Leverett 81].

2.2. Optimization Issues Affected by Microprogrammed Target Machines

The previous section described several features commonly found in horizontal microarchitectures that make them difficult to program. The discussion in this section focuses on the how a horizontal microarchitecture affects the applicability of a number of traditional optimization techniques.

2.2.1. Register allocation

The problem of register allocation arises in compiler optimization because there exist memory hierarchies within a machine architecture. Certain storage locations—usually called registers—are cheaper to access (in time or in space) than others. There may also be machine instructions in which the sources and/or destinations are limited to a certain class of storage locations, or to one location. It is the job of the register allocator to bind program variables and compiler-created variables to storage locations. Sometimes a copy of a storage location is rebound (temporarily) to another storage location to take advantage of access frequency in a particular program segment.

Three features discussed in the previous section affect the problem of register allocation. It was mentioned in Section 2.1.4 that horizontal microarchitectures tend to have a large number of storage classes, which makes register allocation more difficult.

In addition, register allocation can be affected by the higher cost of accessing main memory; the amount of main-memory traffic can easily become a dominating factor when allocating registers for a micromachine. The microcode register allocation schemes designed by Kim and Tan [Kim 79] and DeWitt [DeWitt 76] are based on the premise that main memory traffic should be minimized.

Finally, register allocation can be affected by the difficulty of predicting the cost of a μOp. In order to do a good job allocating registers, it is necessary to balance several costs. For example, if a compiler-created variable contains the result of an intermediate computation, the decision of where to place the variable should take into account costs that include [Leverett 81]:

- The cost of accessing the variable in main memory versus the cost of accessing it in a register.

- The cost of dedicating a register to the variable during the time in which it is live; that is to say, the cost of requiring other variables, which otherwise could have been in a register, to reside in main memory.
- The cost of not storing the value of the variable at all, but rather recomputing its value whenever it is used. This cost may be small if the variable is used *infrequently*.

For vertical architectures, reasonably accurate *estimates* of these costs can be computed by examining the code sequences that perform each task. In a horizontal architecture, however, the estimates of these costs are more difficult to derive.

2.2.2. Flow analysis

Flow analysis is "the transmission of useful relationships from all parts of a program to the places where the information can be of use" [Aho 77]. Such information is necessary in compiler optimizations such as code motion, common subexpression elimination and register allocation [Cocke 70]. Flow analysis may be performed at many stages during the compilation process—in particular, on both source and object code. If a microprogram is being written in a traditional language such as Pascal, flow analysis at the source level will be identical to source-level flow analysis in a traditional compiler. Object-code flow analysis, however, deals with the physical resources of the target machine; the presence of *volatile registers* and *delayed instructions* in a micromachine can make this analysis more difficult.

2.2.2.1. Volatile registers

A *volatile* register is one whose value is implicitly destroyed after a short amount of time. This has an impact on flow analysis because *live* data in a volatile resource must be used or transferred to another *storage location* before the volatile resource loses its data. In a traditional compiler. the data in a storage location can be assumed to be preserved until another instruction explicitly overwrites it. In order to perform flow analysis correctly for a micromachine, it may therefore be necessary to take into consideration the relative distance between μOps, not just the effects of intervening instructions.

2.2.2.2. Delayed instructions

Some microarchitectures have μOps whose effect is delayed for several μIs beyond the execution of the μI in which they occur. This can cause ambiguity in the specification of whether a storage location is *dead* or *live*. Instructions in traditional architectures are (logically) executed serially; between the execution of two instructions, each storage location is in a well defined state. On some micromachines (e.g., PDP-11/40E [Fuller 76]) the full effect of a μI may not be realized before the next μI begins execution.

In this case, there exist two different times at which a storage location may be considered to become *dead*: when the μI that uses the resource has been executed, or when the storage

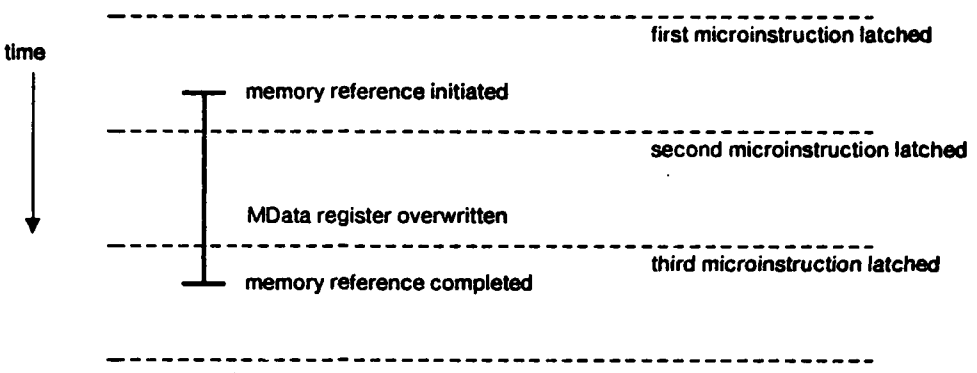

Figure 2-3: Instruction sequence made illegal by delayed execution.

location has been physically read. Consider the example depicted in Figure 2-3. μI 1 contains the μOp that initiates the memory access `MEM[mAddrReg]<-mDataReg`, but because of bus timing, the value of the `mDataReg` is not used until the following μI. If it were assumed that the μIs were executed in a strictly serial fashion, then the μI 2 could contain a μOp which overwrites `mDataReg`, resulting in the wrong data being written to memory.

2.2.3. Local code generation

Although there are many aspects of code optimization, the ability of the code generator to produce high-quality local code is very important [Leverett 79]. Even after other optimization techniques have been applied, there are usually several ways to use an instruction set to produce the same computation. Some macromachines, for example, have addressing modes by which an address computation may be made cheaply; others have special-case instructions for setting a storage location to zero or for incrementing it by one; still others have multiple-action instructions such as *subtract one and branch if zero*. It is important for the code generator to take advantage of such instructions in order to generate code of minimum cost. Because these costs are less predictable for a microarchitecture until the microcode is compacted, we believe that the code generation problem for horizontal machines is more difficult. The coupling of *code generation* and *compaction* is a major topic of this dissertation.

2.2.4. Use of constants

The translation of constants from a source language into machine instructions can also be more difficult for horizontal target architectures. Macroarchitectures tend to have a "standard method" for generating constants (e.g., an *immediate* addressing mode). Compilers for such architectures sometimes also perform transformations such as constant folding and special casing—replacing an addition of the constant "1" with an *increment* instruction, for example.

The production of good microcode can require creativity in generating constants. It is often not feasible to use main memory to store constants needed in the microprogram because it is too expensive to access. Similarly, the specification of a constant in the μI is often expensive because it usually requires a large number of bits that may also be needed for other purposes.

A micromachine may have a collection of hardwired constants. It is often worthwhile to formulate a "difficult" constant in terms of hardwired ones. A micromachine with a shifter and the constant "1" hardwired into it may generate the constant "8" by left-shifting the "1" by three. This type of optimization might be thought of as *constant unfolding*—transforming a constant into a constant expression: the key to producing such a code sequence is the recognition of the fact that "8" can be expressed as "1 leftshift 3". Constant unfolding is discussed further in Section 6.2.5.

2.2.5. Compaction

A horizontal instruction format requires that μOps be *compacted* into μIs. The necessity of compaction is probably the most obvious difference between compilers for vertical and horizontal machines, so it is not surprising that a great deal of attention has been paid to this aspect of microcode optimization. Although progress in the area of μOp compaction will be discussed in detail in Chapter 3, a short analysis of the complexity of the problem and two other issues will be covered in this section.

2.2.5.1. Complexity of the compaction problem

DeWitt [DeWitt 76] proved that the *classical microcode compaction problem* is NP-hard[2] by restricting it to the *unit-execution-time scheduling problem*. Here is an alternate proof, which is based on the NP-hardness of the *graph-coloring problem* [Garey 79].

We restrict the compaction problem by assuming that there are no data dependencies between the μOps. Each μOp is represented by a node in the color-graph; each conflict between two μOps is represented by an arc between two nodes; each μI is represented by a color. The problem of placing μOps into a minimal number of μIs such that no pair of conflicting μOps are in the same μI is isomorphic to the graph-coloring problem: that of coloring nodes with a minimal number of colors such that no pair of connected nodes has the same color.

---

[2] *NP-hard* denotes the class of problems that are at least as hard as any problem in NP. DeWitt claimed that the compaction problem is NP-complete (i.e., both NP-hard and in NP), but did not make the distinction between the *decision problem* and the *optimization problem*. The decision problem, which specifies a constant K and asks whether a given set of μOps can be compacted into a sequence of K or fewer μIs, is certainly in NP. The optimization problem, however, asks for the minimum number of μIs into which the μOps can be compacted; whether or not this is in NP remains an open problem [Garey 79].

This result may be somewhat misleading, however, because in practice, μOps *do* have data interdependencies. In Chapter 7, it is shown that the problem can be solved in polynomial time if the number of registers in the micromachine is bounded; this result is used to argue that the *classical microcode compaction problem* is not properly formulated. The correctly formulated problem is indeed NP-hard.

2.2.5.2. Compaction in the presence of volatile registers

Because a microarchitecture may have *volatile* registers, it is sometimes necessary to force a group of μOps to reside in the same μI. Mallett [Mallett 78] called such a group of μOps a *bundle* and treated each as a single μOp during compaction. Some machines, however, cannot be modeled by single-instruction bundles. If data in a volatile resource is destroyed during the middle of a μI, it may be necessary to place certain μOps a fixed number of μIs from one another [Poe 81]; in other words, a *bundle* might span several μIs. If interblock compaction is performed, it may even be necessary for a bundle to straddle a basic block boundary (i.e., to be divided between two μIs in which either the first contains a branch instruction or the second is a branch target).

2.2.6. Evaluation order determination

Before register allocation is performed, a compiler must determine the order in which program statements are evaluated, and even the order in which subexpressions within an expression are evaluated. If such an ordering is not performed, the register allocation phase will not know the number of compiler-created variables that are necessary at a given point in the program to hold temporary results. The purpose of *evaluation order determination* in optimizing compilers has traditionally been that of minimizing the number of registers required for the evaluation of a given expression or the execution of a given block of code.

For horizontal architectures there is an additional factor that may take precedence over register minimization: the evaluation order puts constraints on how μOps may be compacted. It is therefore possible that a "poor" choice of evaluation order may force μOps to be compacted together that "don't fit very well together." There is thus a circular interdependence among the four tasks, *evaluation order determination*, *register allocation*, *code generation*, and *compaction*:

- *Register allocation* must know the storage requirements for temporary variables, and is therefore dependent on the *evaluation order* of expressions.
- *Code generation* is dependent on *register allocation* because references to different storage classes are likely to be accessed using different sequences of μOps.
- *Compaction* is dependent on *code generation* because μOps cannot be compacted until they are generated.

- It is highly desirable that the task of determining *evaluation order* make use of *compaction* information in ordering expressions so that the final ordering of code "fits together well".

2.2.7. Short-circuit evaluation

Short-circuit evaluation is an optimization often performed by traditional compilers on boolean expressions such as

$$e_1 \text{ or } (e_2 \text{ and } e_3)$$

If the subexpression e_1 is *true*, there is no need to evaluate the e_2 and e_3 (assuming no side-effects); similarly, if e_2 is *false*, it is not necessary to evaluate e_3. It is also *possible* to perform short-circuit evaluation on a numerical expressions—special-casing multiplication by zero, for example. On a traditional machine, however, such an "optimization" is not attractive: program space and execution time would both be increased, except in one case (i.e., first expression evaluates to zero). In a horizontal machine, however, it is possible that μOps to test for the value zero and to perform a conditional branch could be added with no space or speed penalty. The net result in this case would be a program that is occasionally faster—but never slower—than the same program without the optimization. Such an optimization might also be done for other operators or functions, such as *min* when there is a known lower bound on the range of expression values. In short, a horizontal target architecture increases the scope of feasible short-circuit evaluation optimizations.

2.3. Summary

A major problem with generating quality code for a microarchitecture is that it is often difficult to estimate the cost of an instruction (i.e., a μOp). This problem affects aspects of optimization such as register allocation, code generation, and short-circuit evaluation. In addition, other characteristics of microarchitectures, such as volatile resources and the high cost of main memory accesses, may also have an impact on optimization.

Chapter 3
Previous Work

In the past decade, significant progress has been made in the area of compilation for microprogrammed target architectures. This chapter is an overview of the progress in the following areas:

- *Microcode compaction*, the packing of μOps into μIs, attempting to minimize the number of μIs in the program.
- The formulation of a *micromachine model* that covers a large class of microprogrammable machines but is simple enough that reasonably efficient algorithms can be effective.
- The *allocation of registers* to program variables and compiler-created variables in microarchitectures.
- *Microcode generation*, the production of μOps from high-level or intermediate-level programs.

A great deal of effort has been put into the development of effective *compaction algorithms*, particularly in compacting μOps within a basic block. Several authors have concluded that the problem of efficiently—usually optimally—compacting microcode within a basic block is a solved problem [Fisher 81b, Davidson 81]. This conclusion, however, assumes a simple (and usually unrealistic) view of the microarchitecture and the data relationships among μOps, as will be shown in Chapter 7. There also remain unsolved problems in the area of global (i.e., interblock) compaction.

The development of a reasonably general model has also progressed, although there still exist areas in need of further refinement, particularly in the area of micromachine control structures. *Register allocation* and *code generation* have received relatively little attention.

The purpose of this chapter is to give an overview of work in the area of compiler optimization for horizontal target architectures. Its scope is therefore wider than that of the subsequent chapters. We include such topics as register allocation and interblock compaction for completeness.

3.1. Compaction

Because several μOps may be executed during a single μI on a horizontal microarchitecture, it is desirable to compact them as tightly as possible in order to minimize the execution time of—and the space taken by—the microprogram. Most research to date has been limited to the compaction of μOps within a *basic block*—a sequence of μIs with a single entry and exit point. These *intrablock* algorithms are discussed in Section 3.1.1, while research in the area of *interblock* compaction is discussed in Section 3.1.2.

In this section, a simplified model of a microarchitecture is used so that the reader can understand the algorithms without being concerned with low-level machine details; issues regarding more complicated models are discussed in Chapter 5. The simplified model chosen for the discussions in this section is:

> A microprogram is a sequence of μIs, each containing zero or more μOps. The following relations are defined between μOps:
>
> - Two μOps may *conflict*. Conflicting μOps may not be executed concurrently.
> - A μOp may require data that is produced by another μOp. If this is the case, then the former is said to be *data dependent* on the latter.
> - A μOp may destroy data that is required by another μOp. If this is the case, then the former is said to be *data antidependent* on the latter [Banerjee 79].
>
> For the purposes of this discussion we shall use the term *data dependency* when referring to either a dependency or an antidependency, because current compaction algorithms treat them in the same manner; in Chapter 7 it is argued that such treatment is a mistake.
>
> A legal microprogram contains μIs whose μOps satisfy the following constraints:
>
> - Two conflicting μOps may not reside in the same μI.
> - If a μOp is data dependent on another μOp, the former must be placed in a later μI than the latter.
>
> The *classical microcode compaction problem* [Landskov 80] is that of finding a legal microprogram of minimal size.

3.1.1. Compaction within a basic block

With this simplified machine model in mind, let us consider the problem of compacting μOps minimally within a basic block. Because the problem is NP-hard, one might expect all compaction algorithms to consider a large number of μOp orderings; surprisingly, many of the algorithms are linear or near-linear. Three different strategies have been used in addressing this problem:

- Heuristic searches with backtracking [Winston 77]. Each μOp is potentially placed into (and removed from) several different μIs during the compaction.
- Greedy algorithms, which consider the placement of each μOp only once.
- Iterative algorithms, in which each μOp is considered once during each iteration, but which continues compaction until a solution converges.

3.1.1.1. Heuristic searches

One of the earliest published methods for compacting microcode was presented by Yau, Schowe and Tsuchiya [Yau 74]. The algorithm is quite simple, as it performs an exhaustive search with backtracking. For clarity, a nondeterministic version is presented here:

1. Determine data dependencies among μOps based on resource usage.
2. Compute the *data available set*, which is the set of μOps that have not been assigned to a μI, and which are *data dependent* only on μOps which have been already been assigned to a μI.
3. Choose (nondeterministically) a μOp from the *data available set*. If it does not conflict with the current μI, add it to the current μI; otherwise create a new μI and place the μOp there, making the new μI the "current μI".
4. Repeat steps 2 and 3 until all μOps have been assigned to μIs.

Although the algorithm runs in exponential time, and is therefore not practical, it is important historically because many of the current compaction algorithms are based on it.

Yau *et al.* also proposed two pruning methods in order to reduce the search time. The first pruning method only considers in step 1 μOps which do not conflict with the current μI, if any such μOps exist. This guarantees that each μI will be *complete*—a new μI will not be created until it is impossible to add a μOp to the current one. For the simple micromachine model, this pruning method is perfectly reasonable; for more complex machine models, however, such an approach is insufficient (see Chapter 7). The second pruning method, which prunes all but one branch at each node (i.e., backtracking is not performed), is presented in 3.1.1.2.

The compaction algorithm of DeWitt [DeWitt 76] is a variation of the one described above. (His algorithm also performs register allocation, which is being ignored in this discussion.) The μOps are ordered using an evaluation function [Winston 77] that is a weighted sum of the number of μOps in the μI, the number of operands it loads, and the number of new μOps which become *data available* when the μOp is inserted. The search is pruned using upper and lower bounds on the number of μIs in the minimal-length program; these bounds are computed using conflict and data dependency information.

Mallett's experiments [Mallett 78] suggest that both of these algorithms are too slow to perform well in practice. We shall therefore turn our attention to polynomial-time algorithms in the rest of this section.

3.1.1.2. Greedy algorithms

A *greedy algorithm* is an algorithm that generates an approximate solution to an (often) intractable problem by doing a linear-time partial search through the problem space, choosing the locally optimal solution at each point [Horowitz 78]. Current greedy algorithms in the area of microcode compaction fall roughly into two classes. The first class includes versions of Yau's exhaustive algorithm that prune the search tree to one branch at each node. Algorithms in the second class first identify and place "critical" μOps, and then "fill in the holes".

Greedy versions of the exhaustive search have been suggested by Yau *et al.* [Yau 74, Mallett 78], Wood [Wood 79a], and Fisher [Fisher 79]. Except for the method of initially ordering the μOps, all are essentially the same algorithm:

1. Determine data dependencies among μOps based on resource usage.
2. Order the operations according to an *evaluation function*.
3. Compute the *data available set*—the set of μOps that have not been assigned to a μI, and that are *data dependent* only on μOps which have been already been assigned to a μI.
4. Choose the μOp from the *data available set* whose value (as determined by the evaluation function) is the largest among the μOps that do not conflict with the *current* μI, and place it into the *current* μI. If no such μOp exists, create a new (empty) μI—which becomes the current μI—before placing the μOp.
5. Repeat steps 3 and 4 until all μOps have been assigned to μIs.

Yau *et al.* and Wood weight each μOp according to the number of (direct or indirect) descendents in the data dependency graph. Fisher based his choice on experiments that tested twelve ordering strategies, and concluded that ranking the μOps according to their height in the data dependency graph was among the most promising. Poe [Poe 80] is basing his work on Fisher's conclusions and is also using graph height to order the μOps in the compaction process.

A variation of the algorithms above is the *linear pairwise comparisons* algorithm, proposed by Dasgupta and Tartar [Dasgupta 76]. This algorithm differs from the ones previously discussed in that it scans the μOps strictly in order of weight (in this case, *source order*). Data and conflict constraints are used only to bound the placement of μOps, which are placed in the earliest possible μI. Its heavy dependence on the (arbitrary) order of the μOps in the uncompacted source code makes this algorithm rather unattractive.

The *critical path partitioning algorithm* by Tsuchiya and Gonzales [Tsuchiya 74] uses the data dependency graph to identify *critical μOps*—operations which fall on a longest path of the graph. The *critical μOps* are placed in μIs first, with subsequent μOps being placed later;

new μIs are created when necessary. When conflicts require one of two μOps to be delayed, the one with the fewest successors in the dependency graph is chosen. Tokoro *et al.* [Tokoro 81] also use a version of this algorithm.

3.1.1.3. Iterative methods

Gosling's *iterative expansion* compaction algorithm uses a somewhat different approach [Gosling 81]. Instead of beginning with an "empty" basic block and placing μOps until all have been placed, the algorithm begins by placing all μOps into a single μI at the beginning of the block and successively moving them forward until all constraints are satisfied:

```
Compute the dependency relations among the μOps.
Place all μOps into the first μI.
While there is still a (data or conflict) violation do
   For each μOp do
      If this μOp is causing a violation then
         move it to a later time such that it causes no violation, if possible.
```

When there is a choice of two or more μOps to move, the current implementation chooses the one that was later in the initial ordering of μOps. This causes the algorithm to have some of the same weaknesses as the basic pairwise comparisons algorithm of Dasgupta and Tartar [Dasgupta 76]. An obvious extension would be to order the μOps "intelligently" before compaction. The worst-case execution time of the *iterative expansion* algorithm is quadratic in the number of μOps, although its performance is nearly linear in practice.

3.1.2. Compaction involving multiple basic blocks

Interblock compaction algorithms have been the subject of only a limited amount of study. Most techniques that have been considered involve first compacting the basic blocks separately, and then recognizing individual situations in which a μOp (or group of μOps) may be moved between blocks. Dasgupta [Dasgupta 77], Wood [Wood 79b], Poe [Poe 80], and Tokoro *et al.* [Tokoro 81] have proposed such methods. Fisher [Fisher 79, Fisher 81a] developed another technique, *trace scheduling*, that performs both interblock and intrablock compaction simultaneously.

3.1.2.1. Ad hoc methods

Dasgupta [Dasgupta 77] and Wood [Wood 79b] have considered movement of μOps between pairs of basic blocks that surround an *if-then-else* or *case* construct that has no data dependencies involving the μOps being moved.

The strategy being developed by Tokoro *et al.* [Tokoro 78, Tokoro 81] uses a set of rules to determine when μOps may be moved among neighboring blocks; μOps can potentially move long distances when these rules are applied recursively. The usefulness of their algorithm

has yet to be proven, however, because the algorithm does not specify the order in which the interblock motions are attempted or how it is determined whether a *legal* motion is *desirable*. The only data that has been published about the algorithm's performance is based on hand simulations.

Poe [Poe 80] suggested a technique in which each compacted basic block is examined for "holes". When a hole is found, an attempt is made to find a μOp from another block to fill it. As with the algorithm of Tokoro *et al.*, the overall methodology was been reported without experimental results.

3.1.2.2. Trace scheduling

The method that has thus far shown the most promise for interblock compaction is that of *trace scheduling*, which was introduced by Fisher [Fisher 79, Fisher 81a]. A multiblock compaction problem is transformed into a series of basic-block compaction problems in such a way that their solution will result in an effective interblock compaction:

1. Rank each basic block according to the expected number of times it will be executed. Presumably this is determined by "hints" in the source code, or by feedback from execution profiles.
2. Use the rankings to trace a "most common path" through the microcode. Append the basic blocks in this *trace* together, adding artificial arcs to and from branch μOps in the data precedence graph; these arcs represent data dependency relations between μOps that are included in the trace and those in other basic blocks.
3. Compact the trace as if it were a basic block.
4. [Bookkeeping step.] Duplicate all instructions that were moved backward past "join" boundaries in the previous step. Create new basic blocks to hold these μOps and insert these new blocks in front of the respective off-trace blocks that directly join blocks in the trace. This prevents off-trace paths from "losing" μOps.
5. Repeat with other "common" traces, preserving μOps in any basic block which has already been part of a trace.

Trace scheduling has thus far shown the most promise of any the interblock compaction technique. First and foremost, it is the only one that has been implemented to compact microcode successfully. Secondly, the order in which blocks are compacted causes the most frequently executed blocks to be compacted most tightly. Thirdly, it performs intrablock and interblock compaction in parallel, allowing individual blocks to be compacted with a more "global" view. Finally it subsumes many *ad hoc* interblock μOp motions.

3.1.2.3. Compaction involving loops

The horizontal nature of a microarchitecture makes it quite conducive to being programmed in a pipelined fashion. This often results in a large payoff if μOps at the

beginning of a loop can be "rolled back" into the end of the (previous iteration of the) same loop. Consider the three-μI loop in Figure 3-1a. It may be possible for the "load operands" μOp in the first μI to be executed in the last μI of the previous iteration of the loop, thereby reducing the size of the loop by one μI, as is shown in Figure 3-1b.

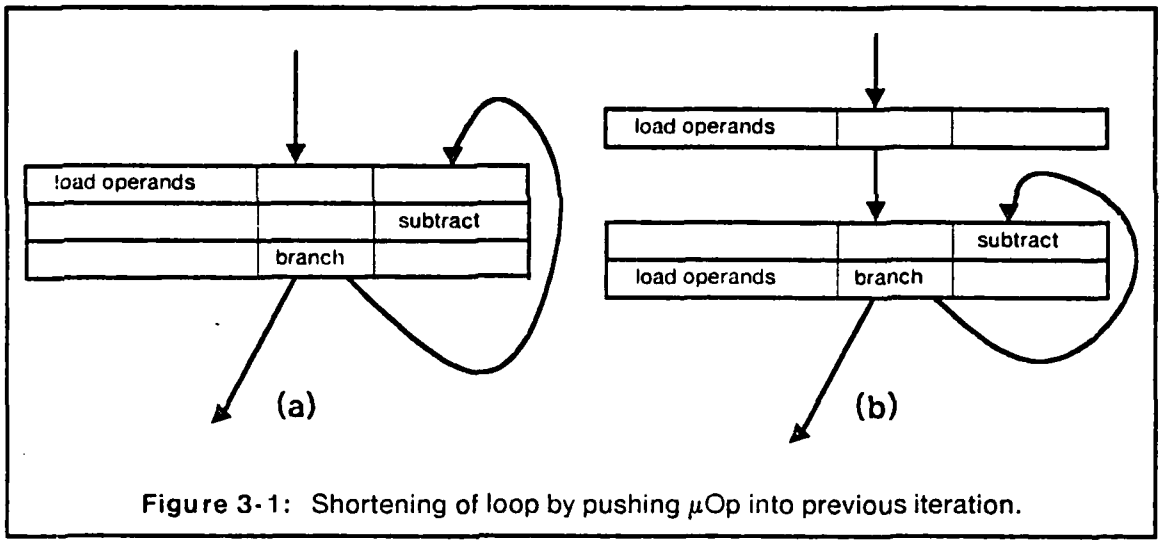

Figure 3-1: Shortening of loop by pushing μOp into previous iteration.

Discussions of loop compaction appear in papers by Fisher [Fisher 79, Fisher 81a] and Poe [Poe 80]. Both consider compacting a loop as a basic block before compacting other blocks; then the loop is treated as a "large μOp", with limited involvement in the rest of the compaction process. There is little discussion anywhere about rolling a loop back into itself.

3.2. Micromachine Models

In the previous section, several microcode compaction algorithms were discussed using an intentionally simple model of a micromachine. This section discusses the development of more realistic machine models.

Because most microcode optimization research has been directed toward the problem of compaction, micromachine models have generally been defined in terms of the two fundamental compaction constraints, μOp conflicts and μOp data dependencies. Other aspects of the micromachine model, such as μOp semantics, have not been addressed to as great an extent. On the other hand, models have been developed that are so general that they encompass almost anything that could be characterized as a digital system [Barbacci 77, Hansen 80]. While general models may be useful for other purposes (e.g., simulation) it is not likely that they characterize microarchitectures in a manner that would be useful for producing optimized microcode; we therefore do not consider them in our discussions.

3.2.1. Conflict determination

Early algorithms simply assumed that there exists a way to determine whether two μOps could be legally placed in a μI [Yau 74]. Although simple, this assumption is still largely valid [Fisher 79] because most current compaction algorithms treat conflict determination as a "black box" subroutine. The remainder of this section is a discussion of the history of μOp conflict models.

Tsuchiya and Gonzales [Tsuchiya 74] pointed out that μOps often conflict because they use a common machine resource. Dasgupta and Tartar [Dasgupta 76] noted that even when two μOps apparently use a single resource incompatibly, that it may still be possible for them to reside in the same word if each uses the resource during a different phase of a polyphase μI cycle. Although the inclusion of polyphase machines in the model affects *conflict determination*, its major effect is in the area of discovering *data dependencies* (see Section 3.2.2).

It is also possible for two apparently independent μOps to conflict because of the format of the μI. DeWitt [DeWitt 76] developed an extensive model of a micromachine control word in which μOps using μI common fields only conflict with one another if they use different values in their common fields; the model of Sint [Sint 81] also takes field values into account. The models of Gosling [Gosling 81] and Fisher [Fisher 81a] do not compare field values; this simpler view is less general, but allows conflicts to be determined using a bit vector. Fisher [Fisher 79] also presented a clever method that allows field values to be encoded as bit vectors that "behave" as simple conflicts.

With the exception of a model extension suggested by Fisher [Fisher 79], the machine models proposed thus far assume that μOp conflict is a binary relation; that is, given any two μOps, it can be determined whether they may reside in the same μI. Hardware considerations, such as fan-out, may make this assumption inaccurate—a bus may exist that may be read by no more into two resources simultaneously; three pairwise compatible μOps that read the bus may cause unstable signals to be generated if executed concurrently. This situation occurs rarely in practice, however, so it is probably not of great importance.

3.2.2. Data dependency considerations

In the previous section, it was suggested that conflict determination can generally be isolated from the rest of the compaction process. Unfortunately, the determination of data dependencies cannot be so isolated; the manner in which data dependencies are modeled can have a profound effect on compaction. Our simple micromachine model states that if a μOp is data-dependent on another μOp, the former must be in a later μI than the latter.

3.2.2.1. Polyphase instructions

Dasgupta and Tartar [Dasgupta 76] included in their model the possibility that a given μOp may only be active during a portion of a μI. On such a machine, it may be possible for two μOps to reside in the same μI even when one is data dependent on the other; this can happen when one μOp executes during an earlier subphase of the μI than the other.

This led to a concept that they called *conditional disjointness* (later called *weak dependence*, and most recently *non-strict dependence*)—a dependency relation in which a μOp may coincide with, but not precede, a μOp on which it is *data dependent*. Previous models had required data dependent μOps to be at least one μI apart.

3.2.2.2. Delays

The model of Mallett [Mallett 78] includes microarchitectures with μOps that require more than one microcycle to complete. Such μOps are rather common; references to main memory, or complex operations like multiplication, often last longer than a single microcycle. Such "long" operations are generally handled in the compaction phase by inserting dummy μOps into the instruction stream [Davidson 81].

3.2.2.3. Volatile registers

Mallett also addressed the issue of *volatile registers* (sometimes called *transitory data resources*) [Mallett 78]. A volatile resource is one which holds its data for only a short period of time, typically one microcycle.

A μOp that reads data from a volatile resource must read it before the data is lost. Mallett therefore introduced the concept of a *bundle*, which is a set of μOps that must reside in the same μI because they pass data via volatile resources. In order to enforce the coresidency restriction, each bundle is treated as a single μOp during compaction.

Unfortunately, *bundles* as defined by Mallett do not successfully model a volatile register whose lifetime extends into the next μI. This subject will be discussed at length in Chapter 7, because it has a non-trivial impact on the compaction problem.

3.2.3. Microoperation semantics

The formalization of μOp semantics has received relatively little attention until recently. This is largely due to fact that most microprogram optimization research has been limited to studying the compaction problem; semantics were modeled only as far as resource usage [Sint 81]. Another reason is probably that the semantics of a μOp—apart from timing—are basically the same as those of an instruction for a traditional machine. Several research efforts in microcode generation have used an existing language, such as ISP [Barbacci 77, Mueller 80a, Mueller 80b, Ulrich 80] or YALLL [Patterson 79, Sint 81], to describe the functional behavior of a μOp.

The recent work of Sint [Sint 81] is directed at both the *code generation* and *compaction* problems and appears to be reasonably general. The usefulness of the model for code generation will be seen as her research progresses.

3.3. Register Allocation

The issue of register allocation for microarchitectures has received a moderate amount of attention. DeWitt and Ma and Lewis base their algorithms on the premise that memory references are extremely expensive; memory-register traffic should thus be minimized at all costs. The effort by Kim and Tan attempts to balance the cost of memory-register transfers with other costs.

DeWitt [DeWitt 76] and Ma and Lewis [Ma 80] each assume that the registers in a microarchitecture are homogeneous, and that uncompacted object code has already been generated. Unbound variables are, of course, named symbolically.

DeWitt performs register allocation in parallel with branch-and-bound compaction. Some of the branches of his heuristic search involve attempting different register/variable bindings, including the insertion of instructions to swap variables between registers and memory. A set of rules is used to prune the search tree, preventing known non-optimal paths from being traversed. Because his experiments were conducted only on small examples, no evidence is presented to indicate that this method is computationally feasible.

Ma and Lewis divide the variables into local/global and dirty/clean classes. If at any point a free register is needed in a basic block, another variable is preempted according to its priority, where the eight priorities are defined by the cartesian product of whether the variable is dirty or clean, local or global, and used or unused in the current basic block. When it is determined that memory-register transfer is necessary, additional μOps are inserted into the object code. Compaction is performed as the final step.

The algorithm of Kim and Tan [Kim 79] includes microarchitectures with heterogeneous registers; allocation is performed among registers classes as well as between the registers and main memory. Costs are balanced between the generation of "optimal" local code, swapping registers in and out of memory, and moving registers between classes within the micromachine. The algorithm itself has four major steps:

- Given the generated object code, with symbolic names for variables, perform flow analysis to determine the portions of the program (if any) where the number of live variables exceeds the number of registers.
- For each such portion of the program, attempt to reduce the number of live variables by applying semantics-preserving transformations.
- If excessive variables still remain after attempting code transformations, insert

load and store instructions to reduce the number of live variables assigned to registers.

- Assign the variables to registers. It may be necessary at this point to insert register-register transfer instructions in order to move variables into the appropriate register class when they are needed—moving a variable involved in an addition into a register which feeds the ALU, for example. An attempt is made to minimize the cost of these transfers. Different combinations of register-register transfer operations and additional load/store instructions are generated, the one with the lowest cost being chosen.

They discuss in detail the methods of cost computation and selection of registers for "spilling" to main memory.

Memory traffic is reduced by flagging portions of the code that require more active variables than there are registers; for such portions of code, a request is made to the code generator to find an alternate sequence which uses fewer registers. If that fails, a variable is swapped out to memory. Because the registers are heterogeneous it may also be necessary to swap data among registers—if the register freed up is of the "wrong" class, for example. An attempt is made to balance the costs of swapping to memory and shuffling registers. Although they do not specify whether the target machine is horizontal, the emphasis on reduction of register-memory traffic and the handling of heterogeneous register classes makes this algorithm an attractive one for microarchitecture register allocation.

3.4. Code Generation

The major goal of microprogram optimization research is the efficient compilation of microcode from a high-level language. Unfortunately, much of this research been limited to the compaction problem, because "horizontalness" is the most striking difference between micro- and macro- architectures. Tokoro *et al.* [Tokoro 78], Wood [Wood 79a], Fisher [Fisher 79, Fisher 81a], and Poe [Poe 80] all presume as a front end to their systems an optimizing compiler that performs all classical compiler optimizations.

This section surveys research efforts that have attempted some form of code generation. None of the systems generate code with the quality of traditional optimizing compilers; many do no optimization at all. If nothing else, this illustrates that there is much work to be done in this area.

Two efforts, the EMPL [DeWitt 76] and Strum [Patterson 76] systems, did not describe their code generation techniques in sufficient detail to be reported here. These two systems are probably not directly relevant to this work as the EMPL system had not been completed at the time it was described, and the Strum code generator made no attempt to optimize code.

3.4.1. Simple code generation systems

As far as could be discerned from their examples, statements in the SIMPL [Ramamoorthy 74] and MDL [Wood 79a] language compilers correspond in a one-to-one manner to μOps in the target machine. The translation process, then, is largely one of matching statements with μOps. Both compilers understand *if* and *while* constructs, and produce branch μOps and labels when control constructs are encountered.

The MUMBLE language [Gosling 81] is largely at the same level as SIMPL and MDL in that program statements correspond to μOps on an almost one-to-one basis. The MUMBLE compiler also contains a graph that represents the data paths of the target machine. If a register-transfer is specified between two registers that are not directly connected, the compiler searches the graph and produces μOps which perform the complete transfer.

Language semantics in the MDIL [Ma 80] and MIMOLA [Marwedel 81] systems are defined in terms of the target machine using a *macro table*. When the compiler encounters a statement in the language, its macro is expanded into machine code.

3.4.2. Code generation with limited optimization

The PL/MP microcompiler [Tan 78] uses a series of templates that associate patterns in the intermediate language with machine language constructs. The templates are ordered in such a way that special cases (e.g., add indirect) are tried before general cases (e.g., add).

Versions of the YALLL compiler [Patterson 79] have been implemented for two different microarchitectures. Simple optimizations are performed, such as the replacement of an "add" μOp with an argument of "1" by an "increment" μOp.

3.4.3. Code synthesis from ISP

Ulrich [Ulrich 80] and Mueller [Mueller 80a, Mueller 80b] have each explored the synthesis of microcode from ISP [Barbacci 77] in a machine-independent fashion using "unconventional" techniques. The ISP statements that were used as "source code" were also quite short. Neither one attempts to produce optimized code or to compact μIs; neither system has yet been shown to be fast enough to be practical.

The system of Ulrich uses symbolic execution techniques. The ISP language is used both as source code and to describe the micromachine semantics. First, a *goal* is set up by symbolically executing the *source* ISP statement. Then different sequences of μOps are symbolically executed until a sequence is found that achieves the *goal*. The current implementation produces correct, albeit inefficient, code.

Mueller attempts to derive microcode using theorem-proving techniques. Micromachine

semantics are specified in a dialect of ISP by defining each μOp in terms of the way it modifies the state of the machine. The first phase of the translation process formulates the source program as a *symbolic assertion*. Next, a theorem-proving process is invoked to verify the existence of a computation which satisfies the assertion. The microprogram is then extracted directly from the proof. At the time of this writing, only a nondeterministic algorithm is implemented; in other words, human intervention is required to guide the program through the search space.

3.5. Summary

Although algorithms for solving the *classical microcode compaction problem* have been developed that appear to perform well in practice, the problem itself does not address the issue of dealing with data antidependencies. Interblock compaction is understood to even a lesser extent, particularly the problem of compacting a loop that "wraps around itself"; details of μOp timing may also complicate the flow analysis necessary to perform interblock compaction.

The development of micromachine models has progressed slowly, but a recent model by Sint [Sint 81] appears to be a reasonable compromise between completeness and utility; because her research effort is in progress, final judgement must be reserved until later.

Research in other phases of optimizing micromachine compilers has progressed much more slowly. Although moderate progress has been made in the area of register allocation, the state of the art in most phases (e.g., code generation) seems to be limited to the techniques used in traditional optimizing compilers.

Chapter 4
Scope of this Research

This chapter defines the set of problems addressed by this dissertation and introduces methods by which the research was performed. First, the central problem—coupling the code generation and compaction phases of compiler for a horizontal microarchitecture—is described. Then three issues are discussed that are closely related to the central problem; these are addressed to a lesser extent in the dissertation. Following that, the scope of this dissertation is delimited by describing related problems that are *not* addressed. Finally, the research methodology is described.

4.1. The Central Problem

This dissertation describes the exploration of three methods by which the *code generation* and *compaction* phases of a compiler for a horizontal target microarchitecture can be coupled. The task of the code generator in an optimizing compiler is that of producing high-quality machine code that preserves program semantics, where quality is defined as a function of time and space costs. As was discussed in Section 2.1.1, these costs are difficult to estimate for horizontal machines until after compaction is performed. The central issue that this dissertation explores is then, *How can compaction information be used to increase the effectiveness of the code generator?*

4.1.1. Some examples

In order to demonstrate that such a problem may arise in a real program, three examples are given. The first involves the addition of a small constant to a register. The second involves generating a test for a loop, while the last involves the interaction of μOp conflicts and a volatile resource.

4.1.1.1. Increment by two

For our first example, consider a situation in which the code generator is required to add the constant "2" to a register on the micromachine sketched in Figure 4-1. An obvious code sequence to perform this operation is one that gates the register onto one input of the ALU

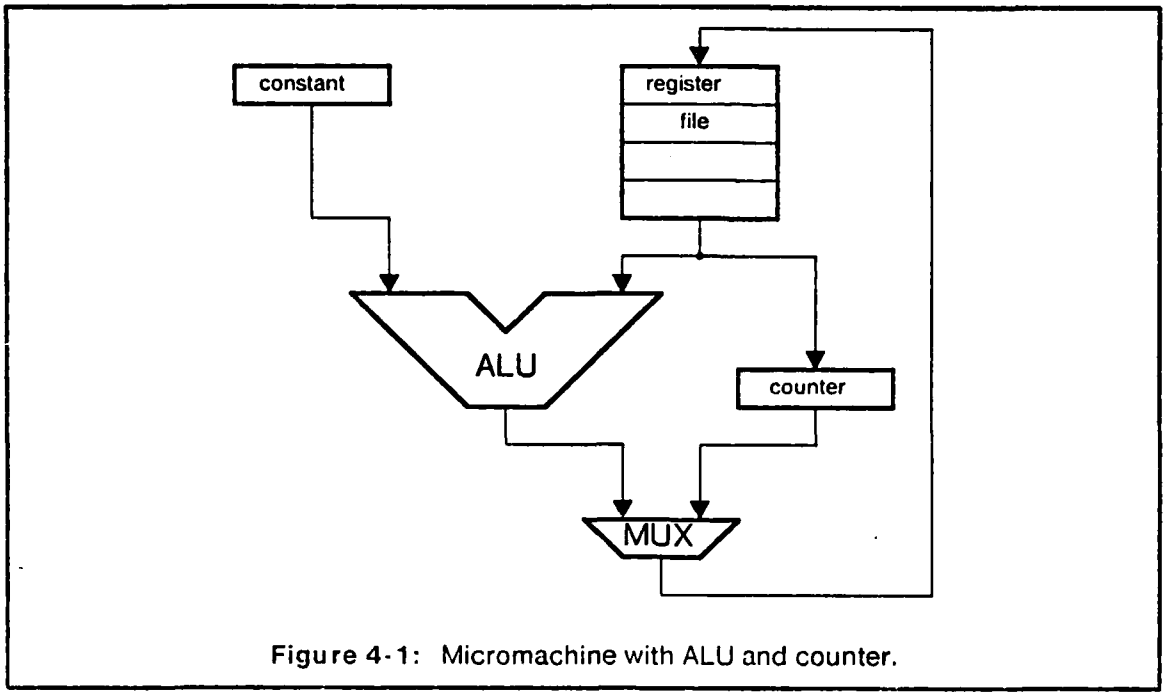

Figure 4-1: Micromachine with ALU and counter.

the constant "2" onto the other, sets the ALU function input to *add*, and then stores the result back into the register. Such a code sequence would probably require one or two μIs, depending on μOp timing.

Another possibility would be to move the register value into the counter, increment the counter twice, and move the counter value back into the register. This sequence would take at least two μIs, and possibly three or four.

In deciding which of these sequences to produce, the code generator might consider the following:

- If surrounding code uses the ALU heavily, but does not use the counter, it is possible that the second sequence can be done for "free"—that is to say, using holes in existing μIs.
- If neither the ALU or counter is overloaded, the first sequence is probably both faster and more compact.
- It is possible that, due to μI field contention, an additional μI or two will have to be inserted in order to produce the constant "2" for the first sequence.
- It is possible that the compaction algorithm can arrange for a prior μOp sequence to leave a constant "2" in a scratch register, making the first sequence more attractive. On the other hand if the constant "−2" could be left in a scratch register, the shortest code sequence might be one in which the ALU performs a subtraction.

4.1.1.2. Loop testing

The second example involves conditional branching on a micromachine in which the computation of branch conditions is overlapped with the fetching of μIs from the control store [Fuller 76, Ousterhout 78, Rosen 79].[3] In such a machine, a conditional branch may require several μcycles to complete; it may be necessary to place the μOps that initiate the conditional branch several μIs before the actual branch is performed. A typical comparison and branch sequence on the Kmap, for example. takes three μIs. During the first μI the ALU inputs are loaded with the values to be compared. The second μI uses the ALU to perform a comparison and to generate condition codes, which are used by the third μI to perform the conditional branch.

Given such an architecture, consider a program containing a loop that is to terminate when the counter reaches the value 50. The code produced in this loop would then include:

1. A μOp that increments a counter.
2. μOps that read the counter (and value 50) into the ALU for comparison.
3. A μOp that branches on the condition generated by the ALU.

The μOps to be compacted would include data dependencies between the μOps in 1 and 2. It is possible, however, that the code for the loop could be compacted more tightly if μOp 1 were somehow allowed move past those in 2. If the code generator and packer were working together, it might be recognized that the order in which 1 and 2 are executed could be reversed, if the the *key* value were changed from 50 to 49, resulting in a semantically equivalent, but shorter, μI sequence. If the remainder of the loop could be compacted even more tightly, this lag might even be two iterations, requiring a comparison with the value 48. The code generator, which is responsible for producing the μOp sequence, does not have enough information before compaction to determine which value to use.

4.1.1.3. Volatile register compensation

As a final example, consider the following simplification of a problem that occurred when the author was writing microcode for the StarOS operating system [Jones 79, Vegdahl 81]. The micromachine, shown in Figure 4-2, has the following hardware constraints:

- The V-register is *volatile*, losing its data at the end of each μI.
- μOps that load the D-register and V-register execute during the first sub-microcycle of the μI; μOps which load the register file, A-register, and F-register execute during the second sub-microcycle. It is thus possible for data to be moved from the A-register to Reg[0] during a single μI, but it takes two μIs to move data from Reg[0] to the V-register.

---

[3] This example involves interblock compaction, which is not directly addressed in this dissertation. It is included as an illustration of the general problem.

- The register file is read and written during the same sub-microcycle, and thus cannot be read and written during the same μI.

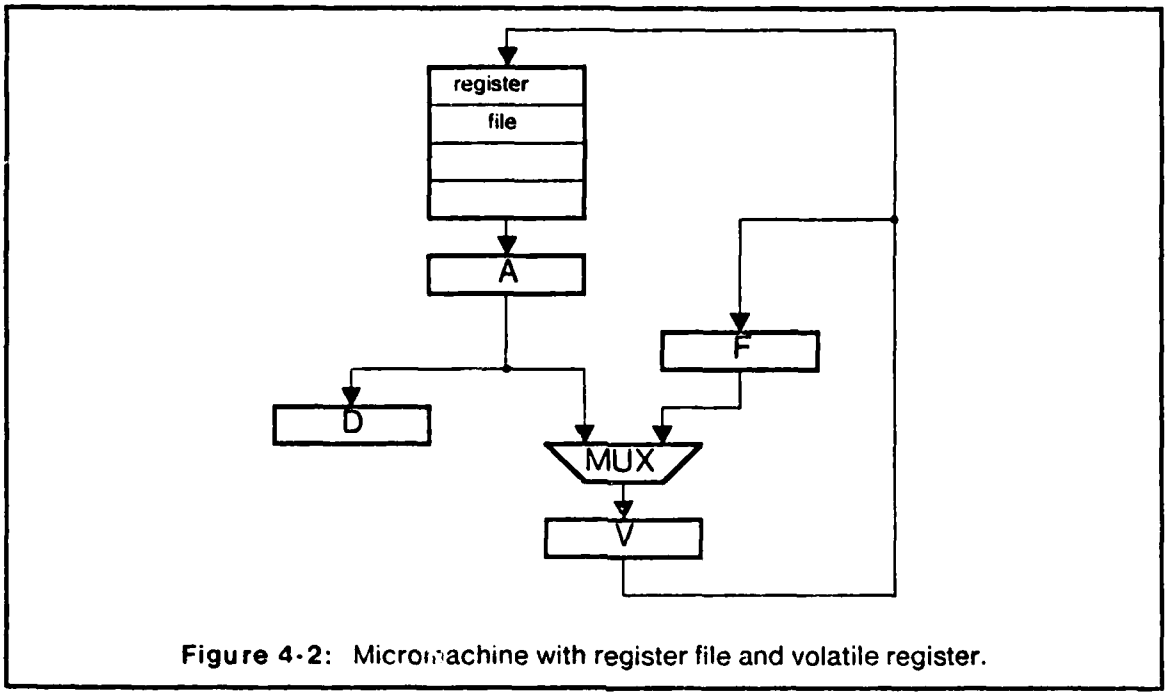

Figure 4-2: Micromachine with register file and volatile register.

With this machine in mind, consider the problem of moving data in the A-register to Reg[1], and the data from Reg[0] into the D-register. The straightforward code sequence for this would be

```
Areg -> Vreg; Vreg -> Reg[1]   move data from Areg to Reg[1]
Reg[0] -> Areg                 move data from Reg[0]
Areg -> Dreg                                   to Dreg
```

This sequence can be compacted into three μIs. The first two μOps must reside in the same μI because the V-register is volatile; the second and third μOps may not reside in the same μI because they both access the register file, while data dependencies require the fourth μOp to follow the third.

Note that if the V-register were not volatile, the second and third μOps could be interchanged, allowing the sequence to be packed into two μIs:

```
Areg -> Vreg; Reg[0] -> Areg
Vreg -> Reg[1]; Areg -> Dreg
```

This can be simulated by using the F-register to hold the data for one cycle:

```
Areg -> Vreg; Vreg -> Freg; Reg[0] -> Areg
Freg -> Vreg; Vreg -> Reg[1]; Areg -> Dreg
```

We see then, an unusual situation in which the execution time of a sequence can be shortened by inserting additional μOps. It is highly doubtful that the code generator in an optimizing compiler would produce this sequence, in which the data traverses an "extraneous" data path, unless compaction were considered.

4.1.2. Summary

The above examples illustrate that it is potentially profitable to couple code generation and compaction. A solution to the first example would involve primarily analysis of resource bottlenecks (ALU, counter), while a solution to the the second depends more on the ability of the two phases to share timing information; the last example has some elements of both.

We do not mean to suggest that this dissertation presents methods for effectively dealing with all three of the above problems; rather, several methods of coupling the phases are explored, leaving the reader the opportunity to judge their strengths and weaknesses. The procedure used in these experiments is outlined in Section 4.4.

4.2. Related Issues

In addition to the problem of coupling compaction and code generation, several other issues relevant to microcode generation are explored here. The development of an adequate micromachine model and code generation and compaction algorithms are necessary prerequisites for the study of the coupling problem. We also explore a technique for generating micromachine constants more intelligently because it appears to be promising.

4.2.1. Machine model

Previous research in the areas of *microcode compaction* and *microcode generation* has produced a number of micromachine models. Unfortunately, the compaction research has produced micromachine models that are too simplified to characterize μOp semantics adequately; similarly, microcode generation research has tended to ignore timing and resource-conflict issues. The machine model presented in Chapter 5 incorporates machine semantics and timing. We do not mean to imply, however, that our model encompasses all microarchitectures; examples of machines that do not completely fit the model are given in Section 4.3.5.

4.2.2. Microcode compaction

Although the problem of microcode compaction has received much attention, we became convinced during the course of this research that further work is needed. The data dependency models used as a basis for current compaction techniques are not adequate. As a result, the solution space is severely restricted; even the exhaustive compaction algorithms consider only a small subset of legal μOp orderings.

In addition, the machine models under which most compaction algorithms have been developed do not allow *volatile* resources to hold data across μI boundaries. When this

feature is introduced to the model, current compaction techniques appear to be inadequate. Examples of this problem are given in Chapter 7.

4.2.3. Constant generation

When translating microprograms that contain constants, the compiler must produce one or more μOps that bring the constant into the micromachine. Possible ways of doing this include:

- Reading the constant in from main memory. This method has a number of shortcomings, not the least of which is that it is likely to be quite slow.
- Reading the constant from the literal field of the μI. This is the most straightforward method of producing a constant in most microarchitectures. It can be somewhat expensive, however, because such fields in the μI tend to be quite wide; one fourth of a 64-bit control word is used to specify a 16-bit constant. Consequently, the literal field is usually *overloaded*, resulting in constraints on the number of μOps that may be executed during a μI in which a constant is specified.
- Producing the constant "creatively". Most microarchitectures have a number of constants "built in" to the machine; these may include masks, small positive and negative integers, and constants that the designers knew would be required for the "primary" application. It may be possible to combine these built-in constants to produce other constants. In the (hand-coded) STAROS microcode [Vegdahl 81], such creative methods were used several dozen times.

This research effort addresses the problem by performing *constant unfolding* during the process of code generation. An attempt is made to express "difficult" constants in terms of "easy" ones in the hope that otherwise unused (or lightly used) resources can be used to remove some of the "constant generation" burden from overloaded fields in the μI.

4.2.4. Code generation

The code generation algorithms in this research are based on the *code-generator generation* algorithms of Cattell [Cattell 78]. Several modifications were made in order to increase the depth of a feasible search. The complexity of the evaluation function for the heuristic search was increased; additionally, the method of ordering the search was modified and a *constant unfolding* mechanism was added.

4.3. Problems Not Addressed

Because of the need to limit the scope of this dissertation, many interesting and important issues relevant to optimized microcode production are not addressed. This section sketches some of the problems that we chose not to address because they did not appear to be as closely related to the phase-coupling problem as those described in the previous section.

4.3.1. Register allocation

The problem of register allocation for traditional compilers has been studied by many researchers, including several that have directed their efforts toward microarchitectures [Kim 79, Ma 80]. While it is the author's belief that there is still much work to be done in the area, it is deemed to be outside the scope of this dissertation. The "variables" given as input to the code generation phase are assumed to be the names of machine resources; when a register is needed for an intermediate value, register allocation is done "on the fly".

4.3.2. Other phase-coupling problems

The reader might have guessed that the code generation and compaction phases are not the only ones that should be coupled in an optimizing microcode compiler. It has been demonstrated, for example, that *register allocation* and *compaction* are another pair of tasks that can benefit from communicating with one another [DeWitt 76]. Similarly, for reasons stated in Chapter 2, *redundant expression elimination* and compaction fall into this category. It also appears that there is a strong interaction between *evaluation order determination* and compaction; this issue is discussed in Chapter 7. Writers of optimizing compilers for traditional machines also face many of the same issues [Leverett 79].

This dissertation focuses on one particular phase-coupling problem in the interest of making the task manageable. It may be possible to generalize this research to some of these other problems at a later time.

4.3.3. Flow analysis

Flow analysis, which can become quite complicated in the presence of unusual timing features (see Section 2.2.2), will be performed only in as much as needed to determine data dependency relationships among μOps for the purpose of compaction.

4.3.4. Interblock compaction

When this project began, we hoped to address the problem of interblock compaction. This topic is outside the scope of the current research effort because of the complex flow analysis it requires, and because unresolved issues in the area of intrablock compaction were discovered.

4.3.5. Machine model

In order to do an effective job at producing and compacting code, we have excluded several microarchitecture characteristics from our model, including:

- Two-level control stores: Nanodata QM-1 [Nanodata 72], MIT Scheme Chip [Holloway 79].
- μIs with variable-length execution times: PDP-11/40E [Fuller 76].
- Subroutines: many micromachines, including the PDP-11/40E [Fuller 76], OM [Johannsen 78], Kmap [Ousterhout 78], and Perq [Rosen 79].

Limitations of the model are discussed in Chapter 5.

4.4. Research Methodology

This section sketches the method by which the issues described in Section 4.1 are explored. We begin with a general discussion of techniques for handling problems of phase coupling, and then present an overview of the three coupling methods that have been explored as part of this research effort.

4.4.1. Coupling methods

As was demonstrated in Section 4.1, when the code generation and compaction phases of a microcode compiler are performed sequentially, many optimizations may be missed. In Section 4.3.2 it was mentioned that there exist phase-coupling problems for compilers in general. This section describes a number of possible techniques for dealing with the problem, of which a subset have been tried as a part of this research effort.

4.4.1.1. Ignoring the problem

Although obvious, the "technique" of doing nothing is probably quite appropriate in a number of situations. A small amount of efficiency gained in the final code may not warrant the additional compiler-writing effort or compile-time [Aho 77]. In addition, an algorithm in which no coupling is done can serve as a benchmark for comparison with other methods.

4.4.1.2. Educated guessing

The method of educated guessing involves performing the phases sequentially, but using heuristics in the first to "guess" what the other phase is going to do; it then performs its task using the "knowledge" it has about the second phase.

This technique has been used by the PQCC group [Leverett 79, Leverett 81] in resolving the coupling problem between the *register allocation* and *code generation* phases of the compiler. The register allocation phase performs an initial code generation in which it predicts the final code that will be generated; this allows it to "know", for example, how many

compiler-created variables will be required for code generation in any given block. This information is then used to make register-assignment decisions.

4.4.1.3. Iteration

Rather than require one phase to make a guess about the behavior of another, it may be appropriate to execute the phases alternately allowing each the opportunity to use the information generated by the previous invocation of the other. This has been shown to be an effective method of dealing with the subphases in object-code optimizers [Wulf 75, Leverett 79].

While this method appears to be appropriate for phases which open up optimization opportunities for one another, it may be quite ineffective in a case where one phase makes a decision that prevents the other phase from performing an optimization; in other words, a poor decision in the first iteration may be propagated into subsequent iterations [Leverett 79].

4.4.1.4. Multiple choices

In situations where one phase detects a potential optimization, but it is the responsibility of another phase to decide whether the optimization is desirable, a scheme might be tried whereby the first phase, rather than performing the optimization(s) it deems best, passes a list of choices to the second. It is the responsibility of the second phase to select the appropriate set of optimizations.

This technique is used in the FLOWAN and DELAY phases of the PQCC project; the existence of such choices is also permitted to a limited extent in the microcode compaction algorithms developed at the University of Southwestern Louisana [Mallett 78, Landskov 80].

4.4.1.5. Performing the phases in parallel

If two phases are highly interrelated, it may be reasonable to incorporate them into the same phase. The Hearsay speech understanding system [Erman 78] used the concept of a *blackboard*, a database common to all phases of the translation process from which any process could read and onto which any could write.

One might also imagine a scenario in which one phase served as a "master" over the other, calling it as a subroutine. A flow analysis phase might be designed as a slave to a number of other modules, each of which requires flow information.

DeWitt [DeWitt 76] designed a microcode compaction and register allocation system in which the two phases called one another recursively. In this case, each phase acted, in some sense, as a master over the other.

4.4.2. Coupling methods to be tested

The research for this dissertation has been carried out in four phases:

- The creation of a micromachine model that is well suited to both *code generation* and *compaction*.
- The development of a machine-independent microcode generation system. The *code-generator generation* algorithms of Cattell [Cattell 78] serve as a basis for the machine-independent *code generation* in our system.
- The extension of the *list scheduling* compaction algorithm of Fisher [Fisher 79] to encompass a more complex micromachine model and a more general notion of data dependency.
- The development and testing of three strategies for coupling the *code generation* and *compaction* phases of the compiler. In the terminology of Section 4.4.1, one is *multiple choice*, one is *iterative*, and one is *parallel*.

The micromachine model is presented in Chapter 5, while code generation and compaction are the subjects of Chapters 6 and 7. The remainder of this chapter briefly describes the three coupling methods, which will be discussed at length in Chapter 8.

4.4.2.1. And/Or

The first coupling technique, which we will subsequently call *And/Or*, falls in the category of "multiple choice" methods listed above. The code generator, rather than producing a single sequence of μOps, produces an *And/Or tree* [Winston 77] from which the compaction phase can choose μOps as it compacts them. An *And/Or tree* is a tree in which each interior node is marked either *And* or *Or*, and the leaf nodes are, in our case, μOps. A solution to a tree consisting of a single leaf is simply the μOp named by the leaf, while a solution to a tree whose root is an *And* node consists of a solution to each of its sons; similarly, a solution to a tree whose root is an *Or* node consists of a solution to any one of its sons.

This coupling method relies on the conjecture that there generally exist only a few μOp sequences that need to be considered; if the code generator can produce them, then the compaction phase has all the information necessary to produce "optimal" code.

This method is used to a limited extent by Mallett [Mallett 78] and the microcode research group at the University of Southwestern Louisana [Davidson 81]. The notion of a *version*—a group of semantically equivalent μOps, one of which must be selected by the compaction phase—was introduced. A version is equivalent to an *And/Or tree* with maximum depth of two (an *And* node at the root and *Or* nodes at the second level) in which all μOps in a *version* must execute during the same μI.

The *And/Or* method is not without its problems. The code generator is complicated by the need to produce multiple "correct" sequences rather than just one, and the compaction

phase must consider an *And/Or tree* rather than a simple code sequence. Chapter 8 discusses these problems and their solutions in detail.

4.4.2.2. Iteration

Consider the following view of microcode optimization:

> A typical block of microcode contains one or more groups of μOps that cause bottlenecks; that is to say, the removal of such a μOp would reduce the total number of μIs required. For example, let us assume that every μI contains a μOp that uses resource *X*. If one such μOp is removed, it may be possible to move the μOps in its μI into surrounding μIs, thereby reducing the code size by one μI. On the other hand, the removal of some other μOp would not reduce the code size because the μOps which use resource *X* would still be required to reside in separate μIs. The code generator, in order to do a good job, should attempt to avoid generating code sequences containing these μOps, preferring μOps that are less likely to be involved in bottlenecks.

The *iteration* method of coupling attempts to produce code that minimizes bottlenecks due to these high-conflict μOps. The code generator uses a table of μOp costs to produce what it believes is optimal code; that is to say, an attempt is made to minimize the sum of the μOp costs. The compaction phase then compacts the μOps into μIs, which are analyzed for bottlenecks. The *cost tables* are updated, increasing the costs of μOps that are involved in bottlenecks; the process is repeated, with the code generator using the updated cost tables.

This method is attractive because it disturbs neither the code generation or compaction phases as such. It involves only the addition of an analysis phase to update the cost tables, and a loop to cause the phases to be repeated. The questions of how to update the tables is discussed in Chapter 8.

4.4.2.3. Squeeze

The third coupling method involves actually performing the phases in parallel. This is achieved by setting the *code generator* as master over the *packer*. Before the code is compacted, constraints are placed on the "shape" on the final code; for example, it might be specified that the final code must be compacted into two μIs, and that the ALU may not be used during the second. The code generator calls the compaction phase whenever it considers a μOp; if the μOp cannot be compacted subject to the initial constraints and the already generated μOps, the code generator searches for alternate code sequences.

With this method, the packer acts as an additional cutoff criterion, pruning the search tree as the code generator attempts to find a code sequence. It is hoped that this method can be extended to the area of producing code for tight loops. The first constraint placed on the final code could be "all μOps must fit into one μI". If that failed to produce a solution, a search could take place with a two-instruction constraint, and so forth. Although this coupling method appears to be quite simple, we encountered a number of problems, which are discussed in Chapter 8.

Chapter 5
Micromachine Model

Before we can produce a machine-independent microcode generator, we must define precisely what we mean by the term *micromachine*. Cattell has noted that the definition of such a class of machines requires tradeoffs between generality and feasibility [Cattell 78]:

> We walk a fine line in making a rigorous definition of a machine in this chapter. On the one hand, we want to include all the machines commonly classified as computers. On the other hand, we want a formal definition that restricts the class of machines enough to make it feasible to automatically generate software. Any useful model must therefore strike a compromise between generality and feasibility.

This chapter defines what a *micromachine* is for our purposes. First the major machine components are discussed informally; then a formal description of the model is presented. Finally, observations are made about the generality and feasibility of the model.

5.1. Overview

The machine model described here is based on that of Cattell, but differs in a number of respects, largely due to differences between macro and micro architectures. The model of storage resources is simpler because horizontal micromachines typically do not have complex addressing modes, which are common in macroarchitectures. The model has been extended, however, to include information about timing and μOp conflicts—that is, the determination of whether two μOps can reside in the same μI.

Our micromachine definition has three major components:

- *Storage resources* are the locations in the machine where data can be stored (e.g., a register) or along which data can be moved (e.g., a bus).
- *Microoperations* (μOps) are the operations available on the machine to move and transform data.
- *Conflict classes* specify which μOps may reside together in a single μI.

Storage resources include busses, latches, register files, and the main memory of the macromachine. The capacity of a storage resource is specified by a *bit length* and a *rank*. The indices of an array storage resource are defined by the μOp semantics.

μOps correspond roughly to the Machine-Operations (M-ops) of Cattell. The semantics of a μOp are defined by an expression, which is represented as a tree; a μOp expression may contain operators, names of machine resources, constants, and constant pattern names. Timing information, which accounts for such features as bus delays and clock phases, is also included.

The text representation of a μOp expression is written in a parenthesized, prefix, LISP-like notation, whose atoms are operators, resources names, and constants. The expression

```
(<- fbus{3 12} (+ (+ areg{2 5} breg{1 5}) 1)))
```

for example, specifies that the *fbus* is to be assigned the value of the sum of *areg*, *breg*, and the constant "1". The numbers in braces specify timing information, which is discussed in 5.2.2.3.

The final component of our machine model is the method of determining whether two μOps can reside in the same μI. Several authors have previously examined this problem [DeWitt 76, Landskov 80], and have included in their models such details as when μOps using a common field might happen to have compatible bit patterns. We have adopted a simpler approach in which the machine description contains a number of *conflict classes*. Typically, a conflict class corresponds to a field in the μI, or to a machine resource. Two μOps that belong to a common conflict class may not reside in the same μI. A μOp may belong to several conflict classes.

Although the μOp conflict model is not as general as it might be, we do not see this as a serious problem. All μOp compaction algorithms we have encountered treat conflict determination as a "black box" subroutine, in which a μOp and a partially-filled μI (or two μOps) are passed in, and a boolean result—"does conflict" or "does not conflict"—is returned. It should thus be relatively easy to extend the model so that it embodies a more general notion of μOp conflicts. During implementation, the "conflict class" model has allowed us to represent conflicts as a bit vector. Additionally, it has allowed us to ignore the explicit bit representation of the μOp, and to produce purely symbolic code.

5.2. Components of the Micromachine

The previous section gave an overview of the micromachine model being used in this research effort. In this section, each component of the model is described more precisely.

5.2.1. Storage resources

The processor state consists of a collection of storage resources. A storage resource is a set of one or more words, each with a fixed number of bits, and is defined in terms of the following components:

- A *name*. This is the alphanumeric string representing the storage resource.
- A *bit length*. This is a positive integer that specifies the word size (in number of bits) of this resource.
- A *rank*. A storage resource consisting of a single word has rank zero. A storage resource that is comprised of more than one word—and therefore must be indexed—has a rank equal to the number of indices that are required to access it. In principle, any multi-word storage resources could be defined to have a rank of one by concatenating its address bits, but we find that allowing multiple indices in our notation is more convenient, and simplifies the heuristic search during code generation.

The size (in words) of a storage resource is never explicitly stated in our model. Instead, it is inferred from the ranges of its indices, as specified in μOp definitions.

There are two resources of rank zero that that the code generator handles in a distinctive manner. The first, called the *micro-address register* (MAR), has special semantics with respect to program execution. The value of this resource at any time determines the μI that is currently being executed. An assignment to this resource causes a branch to be taken, interrupting the default flow of program control; this is discussed further in Section 5.2.4.

The second "special" resource is the *undefined resource*, which is written in the tree-notation as "???". This resource contains a "random" value, and is used to specify that unknown or arbitrary data is assigned to a register or bus. For example, the "and" function in an ALU might specify that the value of the carry out is undefined.

All other storage resources are divided into the two categories *temporary* and *permanent*. A *temporary* resource is one that may be used to compute or store intermediate results—in other words, a value held in such a resource does not need to be preserved during a computation. *Permanent resources*, on the other hand, may not be modified, except as explicitly specified by the source program.

In our model, the instruction memory is assumed to remain unchanged during program execution; its contents may therefore be ignored for the purposes of defining machine state. The contents of the MAR effectively defines the μI that is being executed; the job of the compiler is to bind non-conflicting μOps to potential values of the MAR.

5.2.2. Microoperations

A microoperation (μOp) has the following components:

- A *name*, which is an alphanumeric string that is used to refer to the μOp.
- A *conflict class list*, which is a list of the conflict classes to which this μOp belongs.
- An *expression* that describes the effect of the μOp on the storage resources of the machine.
- Optionally, a list of *constant bindings*, which specify particular constant values for parametrized μOps. For example, a *shift* μOp may require a *shift count* parameter.

We now proceed to describe the *expression* tree; its interior nodes are *operators*, while its leaves are either *constants* or *resource names*.

5.2.2.1. Operators

An operator is represented by a character string and is the leftmost symbol in an expression. The semantics of most operators are defined by axioms (see Section 6.2.1), which are used during code generation to transform the program tree. A few operators, however, have semantics that are explicitly understood by the compiler itself—one may consider the "axioms" for these operators to be represented directly in the compiler code. Examples of such operators are "*if*" (*conditional*), "←" (*assignment*), ";" (*sequencing*), and "loop" (iteration); such operators are understood specially by the compiler because they involve side-effects or control flow. The representation of axioms is described in Section 6.2.1.

In the future, concatenation and shift/rotation operators may be added to the list of operators understood by the compiler. Presently, the compiler does a poor job (i.e., is usually unsuccessful) in compiling code that requires multiple shift and concatenation operations because the evaluation function cannot predict the outcome of such operations to a depth of greater than one. This subject is discussed further in Section 6.4 and in Appendix B.

5.2.2.2. Constants

There exist two types of constants that may be leaf nodes of an expression. The simplest is a *literal constant*, which is an integer value that is represented in the program text in either decimal or octal—an octal number is specified if the leading digit is a zero. For example, the expression

```
(<- areg (and 0777 rbus))
```

specifies that all but the lowest nine bits of *rbus* are to be masked off, the value being stored in *areg*.

The second type of constant that may appear in the program text is a *constant pattern*,

which is represented in the program text as a "**%**" character, followed by an alphanumeric string (e.g., **%wild**). A constant pattern represents a set of constant values, and will match any literal constant that belongs to its set, or another constant pattern of which it is a superset. For example, the expression

```
(<- areg (and %mask rbus))
```

specifies a μOp which may assign to *areg* the *rbus* value "anded" with any value that matches the pattern **%mask**. One special pattern, **%wild**, represents the set of all constants, and will match any literal constant or constant pattern. In the current implementation, each constant pattern is associated with a matching routine that determines whether any particular constant matches the pattern.

When a μOp is first selected by the code generator, the constant patterns in its expression are *unbound*—that is to say, there are no particular values associated with any of its patterns. When the code is compacted, however, specific literal values are associated with each pattern. Thus, a μOp may or may not have a list of constant bindings associated with it, depending on the stage of the compilation process. An unbound μOp is denoted by its name; a bound μOp is denoted by its name, followed by a list of literal values that represent bindings to the constant patterns in its expression. For example, if a μOp with the name *shiftmask* has the expression

```
(<- areg (and %mask (shift %wild breg)))
```

the unbound version of the μOp would be represented by

```
shiftmask
```

while a bound version might be represented by

```
shiftmask 0777700 5
```

where the "0777700" corresponds to **%mask** and the "5" to **%wild**. A μOp whose expression contains no constant patterns is always considered to be bound.

5.2.2.3. Storage resources

A storage resource in an expression is represented by the resource name, timing information, and list of indices whose length is equal to the *rank* of the resource. The general form of a reference to a resource in an expression is

```
<name>{<early time> <late time>}[<index1> <index2> ...]
```

The indices and their surrounding brackets are required for resources whose rank is greater than zero; the number of indices must be equal to the rank of the resource. The value of the index expressions is used to select the particular word in the storage resource that is to be accessed. If a resource has rank zero, the square brackets must be either empty or omitted.

Timing information, which consists of a pair of integers written between braces, is required

for all references to storage resources. The integers refer to times relative to the beginning of μI in which their μOp is placed. Our model assumes that all μI have identical execution times, which, for the purposes of this dissertation, we will (arbitrarily) choose to be ten time units. These time units represent discrete event points during the execution of a μI, and do not necessarily correspond to uniform time intervals.

When a resource name appears as the destination of an assignment statement, the integers in the braces indicate the range of time that the resource will contain valid data. In other cases (i.e., when a resource appears as a source), the integers indicate the time in which the data must be valid in order for the μI to execute properly. For example, the statement

```
(<- areg{3 8} breg{2 4})
```

specifies that if the value of *breg* is stable between times 2 and 4, it will be latched into *areg*, remaining stable there between times 3 and 8. (Remember that all times are relative to the beginning of the μI in which the μOp is placed). It is the responsibility of compiler to guarantee that stability constraints are satisfied.

An asterisk, "*", denotes infinity and is used when an assignment is to be made to a non-volatile resource. Thus, the expression

```
(<- qreg{3 *} breg{2 4})
```

indicates that *qreg* will be assigned the value of *breg* (assuming that breg is stable between times 2 and 4), and will hold that value until the next explicit assignment is made to *qreg*.

If a resource appears in an index expression (i.e., inside square brackets), it is treated as a source even if it appears as part of the destination of an assignment statement. The expression

```
(<- regfile{7 *}[regindex{4 8}] regfile2{6 8}[regidx2{1 7}])
```

indicates a transfer in which the indices must be stable before the source itself.

The specification of timing information in this manner allows a broad range of micromachine timing features to be represented:

- A volatile register whose value remains stable partly into the next μI:

  ```
  (<- areg{5 14} breg {3 6})
  ```

- A resource whose value must be stable even before the μI begins execution (to account for a propagation delay, for example):

  ```
  (<- qreg{6 *} breg{-2 7})
  ```

- A μOp whose execution does not complete until several μIs later:

  ```
  (<- qreg{26 *} (times areg{5 13} breg{5 13}))
  ```

- A resource whose value remains stable for more than one μI, but not forever:

  ```
  (<- areg{5 21} breg {3 6})
  ```

5.2.3. Conflict classes

Conflict classes have two purposes. First, they are used as the basis for determining whether two μOps may reside in the same μI. The rule for determining this is simple: two μOps that have a conflict class in common *may not* reside in the same μI; μOps that have no conflict class in common *may* reside in the same μI.

Second, conflict classes define the cost of each μOp. Each conflict class is assigned an integer cost as part of the micromachine specification; the cost of a μOp is computed by adding together the costs of all conflict classes to which it belongs.

It should be emphasized that costs are defined for the μOps solely for the purpose of guiding the heuristic search during code generation. A μOp has no intrinsic cost of its own; rather it is the μI whose cost is well defined. A μOp is a subset of a μI, but there is no precise way to allocate the cost of the μI over its μOps: at code generation time, it is not known which μIs will contain which μOps. Our goal is to minimize the number of μIs, not necessarily the number of μOps or conflict classes.

There are a number of possible methods for assigning a cost to a particular conflict class; three that might be considered are:

- Assign the value 1 to each conflict class. The cost of a μOp is then the number of conflict classes it is in, which might be a rough measure of the probability of conflicting with another μOp.
- Assign a value to a conflict class that is equal to the number of bits in the μI word it represents. This would cause the cost of a μOp to be the number of bits it requires in the μI.
- Assign a value to a conflict class based on one's expectation that the conflict class will become a bottleneck during compaction.

We have more or less adopted the third approach; this results in the "high-conflict" μOps (based on our estimates at machine-definition time) being considered the most expensive by the code generator.

A final comment about the cost of conflict classes: the *iteration* coupling method modifies the conflict class cost tables in its attempt to induce the code generator to produce better code. Thus, even if the user's estimate of such costs is particularly bad, the compiler has some hope of compensating for it.

5.2.4. Control flow

Micromachines differ greatly in the way conditional branching is performed. The control flow of some micromachines is similar that of a typical macromachine—the *MAR* acts as a program counter and is incremented unless an explicit branch μOp is executed, in which case

a branch may be taken depending on the value of a condition code or machine register. In others, the μI contains one or more explicit destination addresses—the *MAR* is therefore never incremented. The Puma instruction format [Grishman 78] has a *true* and *false* branch address in each μI—a *condition select* field in the μI specifies a condition to test, which is used to select the address of the next μI. The PDP-11/40E and Kmap [Fuller 76, Ousterhout 78] each have a single *next address* field in the μI; conditional branching is performed by *OR*ing condition code values into the lower bits of the *MAR* before the next μI is fetched. Many such schemes cause restrictions on the relative placement of μIs in the control store.

In this dissertation, we wish to avoid issues of placement algorithms in the control store and of characterizing methods by which individual machines perform conditional branching. Such issues have been investigated by others [Fisher 80, Meyers 80, Sint 81] and we believe that most (if not all) of these problems can be handled by a postprocessor to the compiler (e.g., at microassembly time) if code is generated symbolically. We have therefore elected to abstract the conditional branching mechanism by introducing the nondeterministic *flow* operator.

The *flow* operator, unlike most operators, does not represent a single function; rather, it represents the class of injective (i.e., invertible) functions that map integers to integers. When used as the source operand of an assignment statement, the *domain* of the class of functions is the range of its argument, and the *range* of the class of functions is identical to the range of possible values of the destination of the assignment.

For example, let us assume that the *MAR* is a ten-bit register; then the *flow* operator in the expression

```
(<- MAR (flow (> a b)))
```

represents any member of the set of functions that map {0,1} injectively to {0,1, . . . ,1023}.[4] If the functions f_1 through f_4 are defined as

$f_1(0) = 234;\ f_1(1) = 235$
$f_2(0) = 2;\ \ f_2(1) = 1012$
$f_3(0) = 18;\ f_3(1) = 3456$
$f_4(0) = 20;\ f_4(1) = 20$

then functions f_1 and f_2 fall into the class represented by the operator *flow* in the above example, but functions f_3 and f_4 do not; the range of f_3 is not a subset of {0,1, . . . ,1023}, while f_4 is not injective.

The *flow* operator allows conditional execution to be expressed (by assigning to *MAR*), without having to specify the absolute addresses or the low-level details of how the branch is

---

[4] The *greater than* function, represented by the operator ">", returns a boolean result; hence the domain of *flow* in this instance is {0,1}.

effected. The concept that we wish to embody is that the MAR is assigned one of *n* distinct values that depends only on the value of the *flow expression* specified in the μOp. The *n* values are given symbolic names, and it is expected that a postprocessor will bind the symbols to absolute addresses in the control store.

The use of the *flow* operator also allows certain axiomatic simplifications to be easily recognized:

```
(flow (not X))
```

can be simplified to

```
(flow X)
```

representing the fact that the sense of a branch may be reversed. The use of axioms in conjunction with the *flow* operator is discussed more fully in Section 6.2.1.

5.3. Observations about the Model

We now begin a discussion of the generality and feasibility of the model. We first list a number of features found in existing micromachines and discuss reasons for not including them. Then we argue that the model is useful for the task of performing local code generation and compaction.

5.3.1. Limitations of the model

The micromachine model described in this chapter is not entirely general. Part of this is due to the fact that certain aspects of microarchitectures are not relevant to our problem domain. Other micromachine features are excluded or simplified because doing so decreases the difficulty of the implementation (e.g., fewer bookkeeping steps in the algorithm) even though we are aware of no fundamental problems of including them in the model. Still other features are ignored because they *do* introduce fundamental problems, but we felt their inclusion would make the problem too difficult. There are undoubtedly other features of which we are simply unaware, or that will be present only in future micromachines.

5.3.1.1. Conflict classes

One micromachine characteristic that our model does not incorporate is the possibility that the μI bit encodings of two partially-overlapping μOps are compatible; thus two μOps that conflict in our model might be legally representable in the μI. In the Puma [Grishman 78], for example, the literal field overlaps several other functions. If the constant we want to generate happens to have the "right" bits set, however, it is possible to use the literal field in addition to one or more of the other μOps.

As mentioned in Section 5.1, we do not see this as a problem for compaction algorithm to handle, because it treats conflict determination as a "black box" subroutine. By adding the

appropriate information to the data structures and modifying the algorithm to perform the necessary bookkeeping, the compaction algorithm could be modified to handle the more complex model.

Unfortunately, we are also interested in coupling the code generation and compaction phases of the compiler. Some of the algorithms make use of the *conflict class* abstraction in making estimates of the local cost of a μOp. It is not obvious that the coupling algorithms could cope with this extended conflict model without a prohibitive amount of bookkeeping.

5.3.1.2. Timing

Although the model can handle a large class of μI timings, certain timing features are not included. For example, the execution time of a μI on the PDP-11/40E depends on the particular μOps resident in the μI [Fuller 76]; our model assumes that all μIs have identical execution times. In order to extend the model to include such a feature, it might be necessary to express timing information from two different frames of reference. For example, if a main memory reference takes 750 nanoseconds, and μIs take either 250 or 500 nanoseconds, the time between the initiation and completion of a memory reference may be two or three μIs, depending on the particular μOps that are present. This constraint cannot be easily expressed in our notation.

Another assumption made by the model is that a storage resource changes its state instantaneously without going through unstable states. Initially, our solution to this problem was to be "pessimistic" while writing the machine description. If *areg*, during an assignment from *breg*, was unstable from time 2 to time 4, we would write the machine description specifying that the data did not arrive until time 4:

```
(<- areg{4 *} breg{0 4})
```

Unfortunately, this expression does not reflect the fact that the previous value in *areg* could have been destroyed as early as time 2. A compaction algorithm that "trusts" the above expression, and counts on the fact that *areg* will hold its value until time 4, might introduce a timing bug into the program. In retrospect, it would have been better to have three components of timing information, instead of two:

- The earliest time the assignment might cause the old value to be destroyed.
- The earliest time that the new value is guaranteed to be stable.
- The latest time that the new value is guaranteed be stable, assuming no further assignments are made to the resource.

Although our current model assumes that the first two of these are identical, we have opted to leave the system as it is. We felt that making such a change to the model would make a difference in only a few micromachines, and was therefore not worth the effort of modifying the machine descriptions, data structures, I/O routines, and compaction algorithm, even though the modification is trivial conceptually.

A third shortcoming of our model with respect to timing is in the handling of asynchronous logic [Syiek 80, McCreight 80]. The model assumes that each μOp assigns data to a resource at an exact time during the μI. If data along a certain path were not clocked, but rather propagated asynchronously, the timing specification of the μOp would be "whenever the data arrives." The notion of a μOp whose timing is determined by the arrival of its data is not represented in our model. We consider compilation for such machines to be beyond the scope of this dissertation.

5.3.1.3. Dynamic modification of control store

Our model assumes that the control store is read-only and therefore cannot be modified by the program. We do not see this as being overly restrictive because we believe that self-modifying programs should be avoided anyway. Some micromachines, however, allow the control word to be modified after it is read from the control store by allowing additional bits to be *OR*ed into it [Fuller 76]. It may even be the case that this is the only way to address a register file dynamically, or to perform some other task. Our model fails to account for this feature, even though it may be important for some machines. Our philosophy has been to generate code symbolically; the inclusion of this feature would require detailed knowledge of the bit-encodings and placement of μOps. It is still possible to include important special cases (e.g., dynamic register file addressing) by prespecifying to the compiler a sequence of μOps that performs the task.

5.3.1.4. Two-level microcode

We are aware of micromachines that have two levels of control store [Nanodata 72, Holloway 79], often called *microcode* and *nanocode*. There is no way of specifying such machines in our model; we consider such machines to constitute a completely different class of computing engines.

5.3.1.5. Microsubroutines

As the emphasis of our work is on the generation of local microcode, we have chosen to ignore subroutine calls, stack/display management, parameter passing, and other related issues. We believe that there are difficult and important problems in this area, but we consider them to be beyond the scope of this dissertation.

5.3.2. Effectiveness of the model

Although the model excludes a number of micromachine features, we believe that it is quite useful for performing local code generation and compaction for a large class of horizontal micromachines. Cattell [Cattell 78] has already demonstrated that a similar model can be used for generating code for macroarchitectures.

We also believe that the timing and conflict information also facilitates compaction. Our

model choice allows us to represent conflicts as bit-vectors. It can thus be determined whether two μOps may reside together in a μI by performing a bit-mask operation.

The timing constraints between μOps can be determined by subtracting the corresponding components of the source and destination timing information pairs, and then dividing the results by the number of time units in a μI. If μOp *A*:

```
(<- areg{8 15} breg{7 9})
```

produces data for μOp *B*:

```
(<- xreg{3 *} (+ areg{0 3} 1))
```

then the timing constraint between the μOps is determined by considering the timing pairs of the common resource, **areg**. μOp *B* requires **areg** to be valid between times 0 and 3, relative to its μI, while μOp *A* guarantees stability only between times 8 and 15—that is to say from time 8 of the current μI through time 5 of the *next* μI. Thus, the μOps are "timing-compatible" only if μOp *A* executes exactly one μI before μOp *B*. More formally, the range of legal μI offsets between μOps is computed by subtracting corresponding components of the timing pairs, dividing by the number of time units in a μI (which for our purposes is 10), and rounding down or up. Thus, the earliest that μOp *A* can be placed with respect to μOp *B* is

$$\lfloor (dest.early - source.early)/10 \rfloor = \lfloor (0-8)/10 \rfloor = -1$$

or one μI before μOp *B*. Similarly, the latest μOp *A* can be placed is

$$\lceil (dest.late - source.late)/10 \rceil = \lceil (3-15)/10 \rceil = -1$$

or one μI before μOp *B*. Thus the timing information in this example has allowed us to determine that μOps *A* and *B* must be exactly one μI apart.

We believe that this timing model is quite useful. It allows us to compute the relative placement of μOps in μIs, while at the same time allowing a wide range of micromachine timing constraints to be specified.

Chapter 6
Microcode Generation

This chapter describes the heuristic search that performs code generation, which is based on the code-generator generator algorithm of Cattell [Cattell 78]. Because our primary goal is to discover *unusual* code sequences that will compact well under special circumstances, we have rejected the approach of using predefined templates as our only means of generating code, as some microcode compilers have done [Patterson 79, Ma 80]. We wish to use information from the compaction process to increase the power of the code generator. If we were limited to predefined templates, it would be necessary to specify these *unusual* code sequences in advance.

We view the code generator as a testbed for experimenting with methods of coupling *code generation* and *compaction*. This testbed mentality led us to lean very heavily in the direction of flexibility over speed. The model we have selected provides such flexibility by allowing the "intelligence" of the code generator to be increased by adding new axioms.

The remainder of this chapter describes the code generator. We begin with an overview of the code generation algorithm in order to familiarize the reader with the basic concepts. Then, a nondeterministic version of the algorithm is described so that it can be understood without having to consider issues such as ordering and pruning the search. Finally, the problems of making the algorithm deterministic are addressed, and a summary of its effectiveness is given.

6.1. Overview

The code generation algorithm is based on an artificial intelligence technique called *backward chaining means-ends analysis* (MEA) [Winston 77], which presumes an *initial state* (the situation before the solution is applied) and a *goal state* (the desired state). A set of transformation rules is available that transform states to other states. The backward-chaining MEA method may be summarized as follows:

1. The *current state* is initially defined to be the *goal state*.
2. If the *current state* is identical to the *initial state*, then the algorithm terminates.

3. Otherwise, compute the difference between the current state and the initial state, and use this difference to select a transformation rule. Apply the selected transformation rule to compute a new *current state*; then go back to step 2.

For code generation, the *goal state* corresponds to a source-language expression for which a code sequence is desired, the *initial state* to the null expression, and the transformations to *machine instructions* (μOps) and *axioms*. Thus the code generation process is one in which μOps and axioms are successively applied to the goal state until it becomes null. The μOps that are selected during a successful search are those that together satisfy the goal expression.

This process is implemented by two functions, *search* and *transform*. *Search* takes a single argument (a goal expression) and attempts to transform it into the null expression by applying *decompositions* and *μOps*. *Transform* takes two expression arguments and attempts to transform one into the other by applying *axioms*. The two functions call each other recursively, and together implement a depth-first heuristic search with backtracking.

In order make this otherwise exponential algorithm practical, it is necessary to introduce ordering and pruning mechanisms into the search. Selecting the order in which to visit the nodes amounts to ranking the applicable axioms in *transform* and ranking feasible μOps and decompositions in *search*. The most important component of this process is the *evaluation function*, which computes a "distance" between two expressions—that is to say, it estimates the cost of transforming the first expression into the second. The evaluation function is used in conjunction with other heuristics to guide the search.

6.2. Nondeterministic Code Generation Algorithm

This section describes the basic code generation algorithm nondeterministically, ignoring the issues of ordering and pruning the search, which are discussed in Section 6.3. First, the data structures used by the nondeterministic version of the algorithm are described. Then, the algorithm itself is presented, followed by an example. Finally, two extensions to the algorithm—the collection of data dependency information and the use of *constant unfolding axioms*—are discussed.

6.2.1. Data structures

The nondeterministic algorithm makes use of two data structures: a list of μOp definitions, which defines the semantics of each μOp, and a list of axioms, which specifies the transformations that may be applied to expressions during the code generation process. The μOp definitions were presented in Chapter 5 and will not be discussed further here except to say that relevant portions of a μOp's definition are its *name* and the *expression* that specifies its semantics.

An axiom is defined by two expressions that together specify an equivalence-preserving transformation on expression trees. An axiom expression differs from an expression as defined in Chapter 5 in that its leaves may be *axiom parameters* as well as resources or constants. An *axiom parameter* is represented by a "**$**" followed by an positive integer. The *additive commutativity* and *additive identity* axioms, for example, may be represented by

```
(+ $1 $2) :: (+ $2 $1)
```
and
```
$1 :: (+ 0 $1)
```

Whenever a goal is encountered that "matches" the first expression during the search process, it may be replaced with the second expression, where each axiom parameter is replaced by the subexpression that matches it in the first expression.

The axioms in our system are unidirectional—that is to say, the left side is always transformed into the right side, not vice versa. One reason for this is that we allow the pseudo-operator *eval* to be present on the right side of an axiom definition. This operator specifies that constant folding should be attempted when an expression is transformed by an axiom. During the application of an axiom, the *eval* operator specifies that its operand should be replaced with its value whenever it evaluates to a constant; in other cases, the *eval* operator is simply removed. Thus, the associative axiom

```
(+ $1 (+ $2 $3)) :: (+ (eval (+ $1 $2)) $3)
```

transforms

```
(+ 4 (+ 2 areg))
```
into
```
(+ 6 areg)
```

but transforms

```
(+ areg (+ 4 2))
```
into
```
(+ (+ areg 4) 2)
```

A second reason for using unidirectional axioms is the presence of the *flow* operator. Remember from Chapter 5 that this operator is used to specify a flow result (e.g., a branch condition), and thereby represents a whole class of functions. We wish to have axioms that can specify certain properties of *flow*, such as the fact that *identity* and *complementation* satisfy the requirements of the *flow* operator:

```
(flow $1) :: $1
```
and
```
(flow $1) :: (not $1)
```

The converses of these axioms are not true because the left side of an axiom must always be *at least as general* as the right side.

The examples later in this chapter will illustrate the use of axioms in the code generation process. Appendix C lists the axioms used during our experiments.

6.2.2. The algorithm

The code generation algorithm consists of the two mutually recursive functions, *search* and *transform*. The *search* function begins with a *goal* expression, and returns a tree of μOps that satisfies the goal. The *transform* function takes two expressions, *goal* and *current*, and returns a tree of μOps that transforms *goal* into *current*. Typically, *search* is invoked for "statement" expressions (e.g., assignment, conditional, sequencing) and *transform* for arithmetic and logical expressions (e.g., plus, and). We denote a call to *search* by

search: *<goal>*

and a call to *transform* by

transform: *<goal>* **=>** *<current>*

In this discussion, we suppress information about determining the order in which the μOps are executed. Issues regarding the compaction of μOps into μIs are discussed in Section 5.3.2 and in Chapter 7. The collection of control flow and data dependency information—which is used during compaction—is discussed in Section 6.2.4. For the purpose of this discussion, the reader may assume that control flow and data dependency information is automatically generated.

The *search* function *usually* chooses a μOp that is semantically close to the goal, and then invokes *transform* to resolve any differences between the goal and the μOp. In cases where the outermost operator is sequencing (;), conditional (**if**) or repetition (**loop**), a decomposition may be performed instead, resulting in one or more recursive invocations of the *search* function. *Search*, then, is defined as follows:

- A feasible μOp may be chosen whose outermost operator matches the *goal*. *Transform* is then invoked on each operand. When the outermost operator is an assignment, the transformation between the destination operators—but not their indices—is reversed. For example,

  search: (<- w x)

  becomes (after choosing feasible μOp: (← y (+ u z)))

  transform: x => (+ u z)
  transform: y => w *Here we transform the feasible operand into the goal operand, because of the assignment statement.*

  returning the μOp (← y (+ u z)), plus any μOps generated by the two calls to transform.

- If the outermost operator of the *goal* is the sequencing operator, the search may be decomposed into its component parts. For example,

  search: (; (<- a 0) (<- w x))

  becomes

  search: (<- a 0)
  search: (<- w x)

  returning any μOps generated by these two calls.

- If the outermost operator of the *goal* is the conditional operator, the search may be decomposed into its component parts, one of which is the movement of a *flow* result to the *micro-address register* (MAR). For example,

```
search: (if (> a b) (<- x 0) (<- x b))
```

becomes

```
transform: (flow (> a b)) => MAR
search: (<- x 0)
search: (<- x b)
```

returning any μOps generated by these three calls.

- If the outermost operator of the *goal* is an iteration operator, the search may be decomposed into its component parts; again, one of these is the movement of a *flow* result to the *MAR*. For example,

```
search: (loop (<- a (+ a 1)) (> a 10) (<- x (* x 3)))
```

becomes

```
search: (<- a (+ a 1))
transform: (flow (> a b)) => MAR
search: (<  x (* x 3))
```

returning any μOps generated by these three calls. (The **loop** operator defines a generalized looping construct whose operands are executed sequentially; an exit is taken from the loop when the second operand evaluates to *true*.)

The *transform* function transforms one expression into another:

- If the expressions are identical, or *goal* is the *undefined* resource (see Section 5.2.1), as in

```
transform: (+ a b) => (+ a b)
```

return an empty list of μOps.

- If *current* is a constant pattern, and *goal* is a "compatible" literal constant or constant pattern, as in For example,

```
transform: 123 => %wild
```

return an empty list of μOps.

- If both expressions are identical storage resources, but with non-identical indices, *transform* may be called on the indices. For example,

```
transform: regfile[3] => regfile[regindex]
```

becomes

```
transform: 3 => regindex
```

When the call to *transform* had resulted from the matching of assignment statement destinations, the transformation is reversed. This is implemented by setting the *reverse index flag*—a boolean parameter—when the *transform* function is called.

- If *current* is a storage resource, the *fetch decomposition* may be applied:

```
transform: (+ a b) => c
```

becomes

```
search: (<- c (+ a b))
```

- If both operands are expressions with identical outermost operators, *transform* may call itself recursively on corresponding operands. Thus,

```
transform: (+ a (- b c)) => (+ (or x z) y)
```

becomes

```
transform: a => (or x z)
transform: (- b c) => y
```

returning any μOps generated by either call.

- An axiom may be applied to *goal*, followed by a recursive call to transform:

```
transform: x => (+ y z)
```

becomes (after applying the additive identity axiom)

```
transform: (+ 0 x) => (+ y z)
```

Although the *search* and *transform* functions may seem complex, most of this complexity is due either to special knowledge the program has about certain operators, such as *assignment* or *if*. or to special casing on operand type (when the second operand of transform is a constant, for example). During any particular invocation of *search* or *transform*, there are normally only one or two choices that apply.

6.2.3. An example

To illustrate how the different portions of the algorithm work together, let us presume a hypothetical machine with the following μOps:

`AluPlus: (<- ALUoutput (+ aSide bSide))`	*performs an addition in the ALU*
`ASmallNum: (<- aSide %smallNum)`	*sets "A" input of ALU to an integer between 0 and 15*
`BranchZero: (<- MAR (flow (= ALUoutput 0)))`	*performs conditional branch on whether ALU result is zero*
`BReg: (<- bSide reg[regidx])`	*loads "B" input of ALU with a value from the register file*
`ClearCounter: (<- counter 0)`	*sets counter to zero*
`IncCounter: (<- counter (+ counter 1))`	*increments counter*
`SetRegidx: (<- regidx %wild)`	*specifies value of register file index*

and let us assume that the additive identity axiom, `$1 :: (+ 0 $1)` is also available.

```
search: (if (= reg[1] 0)
            (<- counter 0) (<- counter (+ counter 1)))
    apply if decomposition, dividing problem into 3 parts
decide to perform test by getting reg[1] onto ALUoutput
transform: (flow (= reg[1] 0)) => MAR
    apply fetch decomposition
search: (<- MAR (flow (= reg[1] 0)))
    select feasible μOp, BranchZero: (← MAR (flow (= ALUoutput 0)))
transform: (flow (= reg[1] 0)) => (flow (= ALUoutput 0))
    decompose on operand-by-operand basis
transform: (= reg[1] 0) => (= ALUoutput 0)
    decompose on operand-by-operand basis
decide to get reg[1] onto ALUoutput by adding 0
transform: reg[1] => ALUoutput
    apply fetch decomposition
search: (<- ALUoutput reg[1])
    select feasible μOp, AluPlus: (← ALUoutput (+ aSide bSide))
transform: reg[1] => (+ aSide bSide)
    apply additive identity axiom
transform: (+ 0 reg[1]) => (+ aSide bSide)
    decompose on operand-by-operand basis
find code to put 0 onto aSide
transform: 0 => aSide
    apply fetch decomposition
search: (<- aSide 0)
    select feasible μOp, ASmallNum: (← aSide %smallNum)
transform: 0 => %smallNum constants match
find code to put reg[1] onto bSide
transform: reg[1] => bSide
    apply fetch decomposition
search: (<- bSide reg[1])
    select feasible μOp, BReg: (← bSide reg[regidx])
transform: reg[1] => reg[regidx]
    transform indices
transform: 1 => regidx
    apply fetch decomposition
search: (<- regidx 1)
    select feasible μOp, SetRegidx: (← regidx %wild)
transform: 1 => %wild constants match
find code to clear counter
search: (<- counter 0)
    select feasible μOp, ClearCounter: (← counter 0)
find code to increment counter
search: (<- counter (+ counter 1))
    select feasible μOp, IncCounter: (← counter (+ counter 1))
```

Figure 6-1: Example of Code Generation.

Figure 6-1 shows a sequence of calls[5] to *search* and *transform* that produces the following μOps to test `reg[1]`, clearing `counter` if it is zero, and incrementing it otherwise:

```
SetRegIdx 1     set register file index to 1
BReg            read indexed register value onto "B" ALU input
ASmallNum 0     set "A" ALU input to 0
AluPlus         add ALU inputs together
BranchZero      perform conditional branch based on whether sum was 0

Compaction phase and/or postprocessor determines the branch sense,
and inserts any branches necessary after the next two μOps

ClearCounter    control passes here if sum was 0—clear counter
IncCounter      control passes here if sum was not 0—increment counter
```

6.2.4. Data dependency and control flow information

Although the code generation algorithm just described generates code for a large number expressions, we found it necessary to enhance the algorithm in two ways. The first enhancement, discussed in this section, enables the code generator to produce data dependency and control flow information. The second is an extension that increases the power of the algorithm when dealing with constants in the source program, and is discussed in Section 6.2.5.

The algorithm presented in Section 6.2.2 produces a tree of μOps that is semantically equivalent to a given *goal expression*, but does not specify data dependency or control flow information. Thus, it is not necessarily possible to determine the relationships between μOps in the final code by examining the algorithm's output. In order to make this information available later, the algorithm both creates a *flow graph*—a directed graph in which every basic block is represented by a node and each branch between basic blocks by an arcs—and inserts *data dependency links* between μOps.

New nodes in the flow graph are created during the *search* routine whenever an **if** or **loop** decomposition is performed. This is implemented by attaching a label to each μOp identifying the linear block of code into which it is to be placed, and linking together the linear blocks whenever an *if* or *loop* decomposition is performed. We assume that a postprocessor is responsible for binding the labels and μIs to absolute storage locations, and for inserting any unconditional branches necessary to enforce control flow constraints.

Data dependencies between μOps are maintained by associating with each *instance* of a μOp a *copy* of the expression that defines its semantics. Data dependency links are placed between the atomic components (resources and constants) of these expressions in the following situations:

---

[5] In this and later examples, "null" transformations (e.g., areg => areg) are suppressed.

- Whenever an *exact match* occurs during a call to *transform*, data dependency links are created between the respective *atoms* of the two expressions.
- Whenever a *constant match* occurs between two compatible constants during a call to *transform*, a data dependency link is placed between the two constants. Typically, this link specifies a binding between a *literal* and a *constant pattern*. A *pseudo-μOp* representing the literal is passed back as the result of the *transform* function.
- The *sequence decomposition* (when the outermost operator is ";") gives rise to certain implicit data dependencies. For example, there is an implicit dependency involving *b* in the expression

  ```
  (; (<- b 25) (<- a b))
  ```

  Whenever the *search* function applies a sequence decomposition, data dependency links between such resources are created.
- At the end of a call to *search* or *transform*, a transitive closure is performed on data dependencies to account for the fact that the search often involves intermediate expressions.

It is the responsibility of the compaction phase to guarantee that the μOps are compacted in such a way that no data dependencies are violated.

search: (; (<- b_1 25) (<- a_2 b_2))
apply sequence decomposition—this includes setting up a data dependency between b_1 and b_2
search: (<- b_1 25)
select feasible μOp, bWild: (← b_3 %$wild_3$)—we make a copy of the expression, to distinguish this instance of 'b' and %wild from all others that may be generated.
transform: b_3 => b_1
here, we place a data dependency link between the two b's
transform: 25 => %$wild_3$
in this case, we create a pseudo-μOp representing the literal 25, and create a data dependency link to this instance of the pattern %wild
search: (<- a_2 b_2)
select feasible μOp, cB: (← c_4 b_4)
transform: b_2 => b_4
again, just place a data dependency link between the two b's
transform: c_4 => a_2
apply fetch decomposition
search: (<- a_2 c_4)
select feasible μOp, aC: (← a_5 c_5)
transform: c_4 => c_5
place a data dependency link between the two c's
transform: a_5 => a_2
place a data dependency link between the two a's

Figure 6-2: Example of with Search with Data Dependency.

As an example, let us consider the search in Figure 6-2. We have subscripted resource and

pattern names in the example to distinguish between instances of the same atom. It can be seen that data dependencies are placed between references to various patterns and resources as the search progresses; the resulting structure is shown in Figure 6-3.

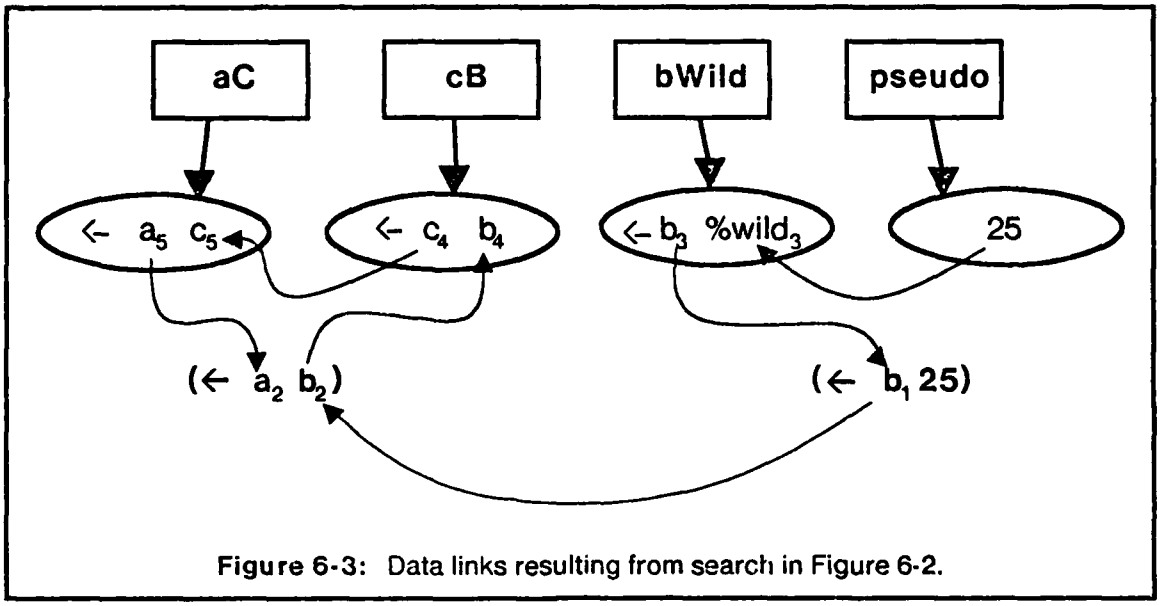

Figure 6-3: Data links resulting from search in Figure 6-2.

At the end of the search, a transitive closure is taken on the data links. This causes all data dependencies between μOps to be expressed as direct links between atoms in their expressions. The resulting structure is shown in Figure 6-4.

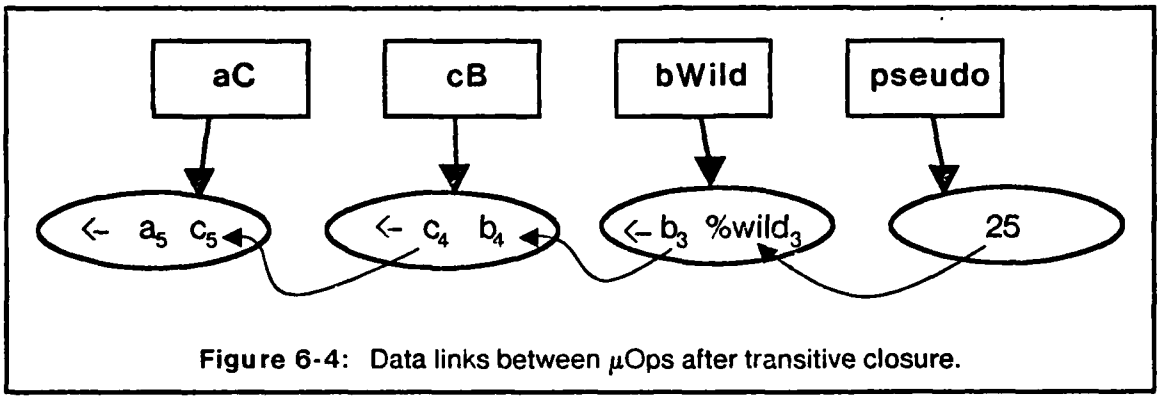

Figure 6-4: Data links between μOps after transitive closure.

Thus, a result tree returned by *search* or *transform* consists of a tree of μOps, each linked to an expression that describes its semantics, where a data dependency between two μOps is represented as a link between atoms of their corresponding expressions.

6.2.5. Constant unfolding

A source program often contains literals (constants) that the compiler must generate during the translation process. A *macromachine* typically has a standard method for generating constants, such as an *immediate* addressing mode. The "standard" method of generating a constant on a horizontal micromachine is often to use a *literal field* in the μI. Such a field, however, is often used for other purposes as well; it is expected that a constant will not be needed during every μI, yet it requires a fairly wide field in the μI to contain the constant—a 32-bit field for a 32-bit machine, for example. This overloading of the literal field leads to μOp conflict restrictions like "a constant cannot be used during the same cycle as a conditional branch" or "a constant cannot be used during a main memory operation."

It is our experience that such restrictions can make the literal field a bottleneck during microcode compaction. We have therefore added to the code generation algorithm a mechanism for discovering methods of generating constants in "unusual" ways by taking advantage of constants that are built into a machine's hardware.

Generating constants intelligently is more difficult for micromachines than for macromachines. The cost of generating a constant on a macromachine is typically no more than one word of code (space) and one memory reference (time); there is thus a fairly tight bound on the complexity of any solution that is better. For micromachines, however, it is possible for an arbitrarily complex solution to be *optimal* in a given situation, as long as its μOps fill "holes" in μIs that would otherwise be vacant.

The original goal of our research in this area was that of building a mechanism that would allow code sequences to be generated that would avoid using the *literal* field of a μI. We were surprised to discover that this mechanism is capable of discovering optimizations beyond those originally envisioned.

6.2.5.1. The basic mechanism

The basic mechanism for generating constants is the application of *constant unfolding* axioms during the search. A constant unfolding axiom replaces a constant by a constant expression of equal value. The goal is to make use of constants that are hard-wired into the micromachine, replacing difficult-to-generate constants with expressions involving only hard-wired constants. Constant unfolding axioms are applied during the *transform* function in the same way other axioms are applied.

As an example, let us consider the problem of adding the value "8" to a register R, given a micromachine in which "masking" constants (e.g. 0, 1, 3, 7, 15) are built into the machine. The straightforward method of performing the operation would be to generate the constant "expensively" (using the literal field), gating it to one input of the ALU, and to place the value of R at the other input.

```
decide to compute result by adding 8 on bSide, R on aside, with carry 0
search: (<- R (+ R 8))
    select μOp: (← R ALUoutput)
 transform: (+ R 8) => ALUoutput
     apply fetch decomposition
  search: (<- ALUoutput (+ R 8))
      select μOp: (← ALUoutput (+ (+ aSide bSide) carryIn))
   transform: (+ R 8) => (+ (+ aSide bSide) carryIn)
       apply additive identity axiom
    transform: (+ (+ R 8) 0) => (+ (+ aSide bSide) carryIn)
        decompose on operand-by-operand basis
     select code to put 0 in carryIn, R on aSide
     transform: 0 => carryIn
         apply fetch decomposition
      search (<- carryIn 0)
          select μOp: (← carryIn 0)
     transform: (+ R 8) => (+ aSide bSide)
          decompose on operand-by-operand basis
      transform: R => aSide
          apply fetch decomposition
       search: (<- aSide R)
           select μOp: (← aSide R)
     decide to put 8 on bSide by adding 1 and 7
     transform: 8 => bSide
         apply constant unfolding axiom
      transform: (+ 1 7) => bSide
          apply fetch decomposition
       search: (<- bSide (+ 1 7))
           select μOp: (← ALUoutput (+ 1 bSide))
        transform: ALUoutput => bSide
            apply fetch decomposition
         search: (<- bSide ALUoutput)
             select μOp: (← bSide ALUoutput)
        select code to get 7 onto bSide
        transform: 7 => bSide
            apply fetch decomposition
         search: (<- bSide 7)
             select μOp: (← bSide %MaskConstant)
          transform: 7 => %MaskConstant
              7 matches the %MaskConstant pattern
```

Figure 6-5: Search with constant unfolding.

Figure 6-5 shows how constant unfolding can be used to generate this alternate code sequence:

```
(<- bSide %MaskConstant) 7   put constant 7 on B input to ALU
(<- ALUoutput (+ 1 bSide))   increment the 7, getting 8 on ALUoutput
(<- bSide ALUoutput)         swing the 8 back to the B input
(<- aSide R)                 place value of register R in A input
(<- carryIn 0)               set carry-in value to 0
(<- ALUoutput (+ (+ aSide bSide) carryIn))
                             use ALU again, computing R + 8 + 0
(<- R ALUoutput)             store result back in register R
```

This sequence does not use the literal field of any μI. The ALU, however, is used during two cycles.

6.2.5.2. An extension

The above method can be useful when it is necessary to produce a constant explicitly. The mechanism can be extended, however, by applying its axioms to subexpressions. This can allow a constant in the source program to be unfolded and combined with other expressions, often resulting a code sequence in which the constant is never explicitly generated during execution. Figure 6-6 shows how the application of constant unfolding at the subexpression level can improve the code sequence generated in Figure 6-5:

```
(<- bSide %MaskConstant) 7   place constant 7 onto B ALU input
(<- aSide R)                 place value of register R onto A ALU input
(<- carryIn 1)               set carryIn to 1
(<- ALUoutput (+ (+ aSide bSide)) carryIn)
                             compute value R + 7 + 1 in ALU
(<- R ALUoutput)             store value back into register R
```

This sequence not only avoids using the literal field, but also uses the ALU during only one μI. This is a result of performing *constant unfolding* at the subexpression level so that the associativity axiom can bring the "1" portion of the unfolded constant into a position where it can be matched with "carryIn". This follows a pattern that will also be seen in the remaining examples:

- First, a constant unfolding axiom is applied to a subexpression.
- Then, another axiom—usually associative or distributive—is applied to the entire expression, causing portions of the unfolded constant to be combined with other portions of the expression.
- The portions of the unfolded constant are matched with different (and perhaps distant) μOps, often generating a code sequence in which the original constant is never generated explicitly.

As another example, consider the problem of adding the constant "2" to a register on a machine that has a counter. Again, the straightforward method of doing this would be to use the literal field of the μI to generate a "2", and to use the ALU to perform the addition. An alternate method would be to load the value into the counter and increment it twice, as can be seen in Figure 6-7. The resulting code,

```
decide to use ALU to perform addition
search: (<- R (+ R 8))
    select μOp: (← R ALUoutput)
 transform: (+ R 8) => ALUoutput
     apply fetch decomposition
  search: (<- ALUoutput (+ R 8))
      select μOp: (← ALUoutput (+ (+ aSide bSide) carryIn))
   unfold constant, and use associativity to match up corresponding parts
   transform: (+ R 8) => (+ (+ aSide bSide) carryIn)
       apply constant unfolding axiom
    transform: (+ R (+ 7 1)) => (+ (+ aSide bSide) carryIn)
        apply additive associativity axiom
     transform: (+ (+ R 7) 1)) => (+ (+ aSide bSide) carryIn)
         decompose on operand-by-operand basis
      find μOps to load ALU inputs
      transform: 1 => carryIn
          apply fetch decomposition
       search (<- carryIn 1)
           select μOp: (← carryIn 1)
      transform: (+ R 7) => (+ aSide bSide)
           decompose on operand-by-operand basis
       transform: R => aSide
           apply fetch decomposition
        search: (<- aSide R)
           select μOp: (← aSide R)
       transform: 7 => bSide
           apply fetch decomposition
        search: (<- bSide 7)
           select μOp: (← bSide %MaskConstant)
         transform: 7 => %MaskConstant
             7 matches the %MaskConstant pattern
```

Figure 6-6: Search with constant unfolding on a subexpression.

```
(<- counter R)
(<- counter (+ counter 1))
(<- counter (+ counter 1))
(<- R counter)
```

completely avoids using the ALU. Again, performing constant unfolding at the subexpression level is critical in discovering the code sequence.

Another example of the use of constant unfolding in discovering nonstandard methods of generating constants is the problem of performing a masking operation.[6] Let us hypothesize a machine which has built-in masking constants of the form $(2^n - 1)$ and their complements—in other words, the (binary) constants 0, 1, 11, 111, etc. and 11111111, 11111110, 11111100, etc. Thus any number of high (or low) bits may be masked off using an "easy-to-generate" constant. Let us then consider the problem of generating the expression:

[6] In this example, binary notation is used for clarity. 8-bit data is assumed so that binary constants can be written in reasonable space.

```
decide to use counter to increment
search: (<- R (+ R 2))
    select μOp: (← counter (+ counter 1))
 transform: counter => R
     apply fetch decomposition
  search: (<- R counter)
      select μOp: (← R counter)
unfold constant, and match "outermost" 1
transform: (+ R 2) => (+ counter 1)
     apply constant unfolding axiom
  transform: (+ R (+ 1 1)) => (+ counter 1)
      apply additive associativity axiom
   transform: (+ (+ R 1) 1)) => (+ counter 1)
       decompose on operand-by-operand basis
    find code to increment again, and to store result
    transform: (+ R 1) => counter
        apply fetch decomposition
     search: (<- counter (+ R 1))
         select μOp: (← counter (+ counter 1))
      transform: (+ R 1) => (+ counter 1)
          decompose on operand-by-operand basis
       transform: R => counter
           apply fetch decomposition
        search: (<- counter R)
            select μOp: (← counter R)
```

Figure 6-7: Constant unfolding used to avoid ALU μOps.

```
(and 00111000 reg)
```

In this case, the constant 00111000 may be unfolded in three ways, each unfolding resulting in a different code sequence. It may be expressed as the bit product of two masks and then transformed by an associativity axiom,

```
(and 00111000 reg) => (apply constant unfolding)
(and (and 11111000 00111111) reg) => (apply associativity)
(and 11111000 (and 00111111 reg))
```

resulting in a code sequence in which *reg* is first masked with 00111111 and then by 11111000. Alternatively, we may express the constant as a rotated mask and then apply a distributive axiom,

```
(and 00111000 reg) => (apply constant unfolding)
(and (rotLeft 3 00000111) reg) => (apply distributive law)
(rotLeft 3 (and 00000111 (rotRight 3 reg)))
```

resulting in a code sequence in which *reg* is rotated right by 3, masked and rotated back. Similarly, we may express the constant as a mask rotated in the opposite direction and apply a distributive axiom,

```
(and 00111000 reg) => (apply constant unfolding)
(and (rotRight 2 11100000) reg) => (apply distributive law)
(rotRight 2 (and 11100000 (rotLeft 2 reg)))
```

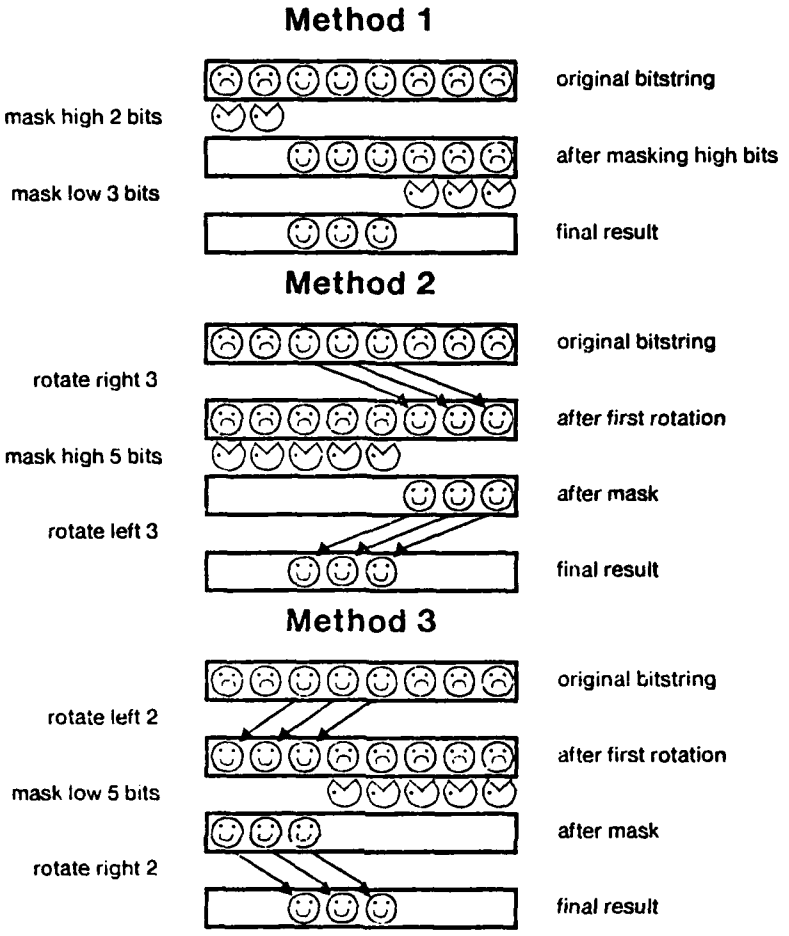

Figure 6-8: Three methods of performing a masking operation.

causing *reg* to be rotated left by 2, masked and rotated back. Diagrams illustrating the three code sequences discussed for this problem are shown in Figure 6-8.

Our final example illustrates the use of constant unfolding in conjunction with a distributive law and strength reduction, in "discovering" that a multiplication by the constant "3" is equivalent to a shift and add:

```
(* 3 x) => (apply constant unfolding)
(* (+ 1 2) x) => (apply distributive law)
(+ (* 1 x) (* 2 x)) => (apply identity and strength reduction axioms)
(+ x (shiftLeft 1 x))
```

6.2.5.3. An implementation note

We have found that the analysis necessary for doing an effective job of unfolding constants has been difficult to formalize; such axioms can be expressed in the same way that other axioms are expressed, but it is sometimes necessary tc introduce new axiom parameters on right side of the definition. This would make it necessary for the axiom mechanism to make a

nondeterministic choice for unbound variables. For example, an axiom that unfolds a constant into a sum of two others might be expressed as:

```
$1 :: (+ $2 (eval (- $1 $2)))
```

When the constant is unfolded, a value must be chosen for **$2**.

For this reason, the current implementation requires that the set of constant unfolding axioms be represented by a routine in the code itself. This routine takes two operands: if the first is a constant, it returns a list of constant expressions whose values are identical to the first operand, but that are "good candidates" for matching the second operand. If the first operand is an expression, it attempts to unfold any constant suboperands, and returns a list of expressions that are equivalent to the first operand, but with one of the constant suboperands unfolded. Currently, it is necessary to write for for each target microarchitecture a new routine that "knows" about generating constants for that particular architecture. We hope that methods for making such analysis machine-independent can be developed in the future.

6.2.5.4. Summary

We have found that constant unfolding axioms lead to discovering code sequences that generate constants in non-standard ways. In particular, their application at the subexpression level is a quite powerful, and can lead to the discovery of code sequences that could not otherwise be discovered by the code generator.

We have not attempted to apply constant unfolding axioms to subexpressions whose depth is greater than one. According to our experience, this is not necessary, as we have never encountered a situation in which the unfolding of a constant at a greater depth would have increased the effectiveness of the code generator.

6.2.6. Summary

In order to make the formalism of Cattell suitable for micromachine target architectures we have modified his algorithms to fit our machine model. In addition, we have added mechanisms for keeping track of data dependencies between μOps, and for performing *constant unfolding*.

We are now ready to present the final version of the nondeterministic code generation algorithm:

Search(goal) =

- A feasible μOp may be chosen whose outermost operator matches the *goal*. *Transform* is then invoked on an operand-by-operand basis, returning all μOps from all such calls to *transform*. If the outermost operator is an assignment, the transformation between the destination operators is reversed, with the *reverse index flag* being set.

- If the outermost operator of the *goal* is a sequencing operator, the search may be decomposed into its component parts, and data dependency links added between certain references to resources in the original expression.
- If the outermost operator of the *goal* is a conditional or iteration, the search may be decomposed into its component parts, one of which is the movement of a *flow result* to the *MAR*. New flow graph nodes and links are also generated.

Transform(goal, current) =

- If the operands are identical constants or resources, place a data dependency link between *goal* and *current*; the operands are identical expressions recursively call *transform* on corresponding suboperands. Return an empty list, signifying that no μOps are necessary to transform the first operand into the other.
- If *current* is a constant pattern, and *goal* is a "compatible" literal constant or constant pattern, place a data dependency link between *goal* and *current*, and create and return a pseudo-μOp (as defined in Section 6.2.4) whose operand is *goal*.
- If both expressions are identical storage resources with non-identical indices, *transform* may be applied to the indices; if the call had been made with the *reverse index flag*, the sense of the transformation is reversed.
- If *current* is a storage resource, the fetch decomposition may be applied, resulting in a call of the form:

```
search: (<- current goal)
```

- If both operands are expressions with identical outermost operators, *transform* may call itself recursively on an operand-by-operand basis, returning all μOps generated by any of the calls.
- An axiom may be applied to *goal*, followed by a recursive call to *transform* the modified *goal* into *current*.
- If *goal* or one of its suboperands is a constant, a constant unfolding axiom may be applied to *goal*, followed by a recursive call to *transform* the modified *goal* into *current*.

6.3. Deterministic Code Generation Algorithm

Because the nondeterministic algorithm requires exponential time when run on a uniprocessor, it is necessary to limit the number of nodes that are examined during the search. Initially, we considered using heuristics similar to those used by Cattell [Cattell 78]. In his system, a predetermined integer, the *depth limit*, specified the maximum depth in terms of number of recursive calls to the *search* and *transform* functions. No other pruning or ordering was performed on axiom applications. The feasible instructions were ordered by performing some simple expression comparisons, and were pruned using a *breadth limit*—an upper bound on the total number of nodes searched at or below any given level in the search tree.

Our experiments have convinced us that the mechanisms developed by Cattell are not sufficient for generating microcode. Our heuristic searches tend to be deeper than his, because our code generator must produce longer instruction sequences. This is partially due to the difference in machine architectures; our algorithm must discover longer code sequences because our "instructions" are μOps, each of which tend to change the state of the machine in only a small (micro!) way.

Another reason that our searches tend to be longer is that our task is that of a *code generator*, while his was that of a *code-generator generator*. Input to his algorithm tends to be a set of reasonably simple expressions, resulting in code sequences of one to three instructions in length. Input to our system can be a block of code, sometimes requiring the production of a dozen or more μOps.

The requirement of a greater search depth has its obvious drawbacks. Because the time complexity is exponential in search depth, we must either accept the exponential time increase or develop a searching strategy that performs more pruning. Experiments have convinced us that the former approach is not feasible; we have therefore introduced a more complex searching strategy and evaluation function.

The remainder of this section discusses the important issues that arose as we implemented the code generation system, and outlines our solutions. A detailed discussion of the deterministic algorithm may be found in Appendix A; details of the evaluation function algorithm are given in Appendix B.

6.3.1. Search depth

One of the major questions we faced in building the system was that of defining what was meant by the term *search depth*. In Cattell's system, depth is defined simply by the number of recursive calls to the *search* and *transform* functions. In our system, however, it is sometimes necessary for the depth of the search (by this definition) to reach 20 or more; we certainly cannot afford to examine all nodes in the search tree at that depth!

Instead we define the depth of a node in the search tree to be the sum of the costs of the μOps that lie along the path that connects it with the root. A search may therefore be quite deep (in the number of calls) as long as it selects only inexpensive μOps.

In order to approximate a breadth-first search—which has a number of attractive properties—without incurring the storage costs that are typically associated with a breadth-first search, we use the iterative deepening [Slate 77]. When a search is started, it is passed a "cutoff" value that defines the depth beyond which it is not allowed to examine nodes; this is implemented by reducing the cutoff whenever a μOp is selected during the search. If the

search terminates without having found a solution, the cutoff is increased and the search is retried, the process being repeated until a successful solution is found.

When a search is passed a particular cutoff value, our intention is that the search will find a solution only if one exists whose total cost is not greater than the cutoff. Unfortunately, a search can be partitioned into subsearches (e.g., operand-by-operand decomposition), leading to a situation where the total cost can exceed the cutoff. In order to remedy this situation, the cutoff value is divided among the subsearches whenever such an occasion arises. Our experiments suggest that the search is most effective when such an allocation heavily favors the subsearches that are deemed (by the evaluation function) likely to be the most expensive.

6.3.2. Pruning and ordering the search

The evaluation function (see Section 6.3.3) is used as the primary method of pruning the search and determining the order in which nodes are examined. A path along the search tree is pruned whenever its cost—as estimated by the evaluation function—exceeds the cutoff value; nodes in the search tree and examined in ascending order of cost, again as estimated by the evaluation function.

A small number of other pruning mechanisms are also employed, primarily because experiments indicated that the evaluation function often allows axioms to be applied so profusely that the search explodes exponentially. Most of these heuristics are ones that require primary operators or destinations (for assignment statements) to match; one heuristic limits to three the number of axioms that may be applied at any node in the search tree.

We also introduced a caching mechanism that has proven to be useful in pruning the search: if a particular search has already failed at the current depth, the path is aborted immediately. The caching mechanism also acts as a *memo function* [Michie 68]: a previously successful search need not be repeated.

6.3.3. The evaluation function

The purpose of the evaluation function is to give an estimate of cost of transforming the machine from one state into another. Its parameters are two expressions, a *goal expression* and a *current expression*. The evaluation function recursively compares various subexpressions of the *goal* and *current* expressions, and uses "distance tables"—generated from the machine definition and axioms—to arrive at the final estimate. An extensive description of the evaluation function is given in Appendix B.

6.4. Results

We conclude from our experiments that the system does a reasonably good job of producing microcode for source expressions that only require data to be moved along busses and through ALU's and masks, and constants to be generated. We were particularly pleased to find that it performed quite well on a subset of the Puma microarchitecture the first time that we tried it, and even discovered one code sequence that was better than we had anticipated. In addition we feel the "discovery" that incrementing a counter three times is equivalent to adding the constant "3" was impressive.

Our system is able to perform searches that are much deeper than those performed by the prototype implemented by Cattell, but is also slower. It has produced a successful search to a depth of 28 calls to *search* or *transform*, and has applied axioms in a successful search to a depth of 11. Cattell's system, which used a much simpler evaluation function, searched to maximums of 8 and 3 respectively. We by do not mean to imply that our system will always be successful in searches as deep at 28 and 11; more typical search depths are 13 and 4. As far as execution time is concerned, Cattell's system, which was written in SAIL, typically examines 200 nodes in the search tree per second when running on a DEC KL-10 [Bell 78]; our system, which was written in Berkeley Pascal, examines about 30 nodes per second when running on a DEC VAX/11-780 [Strecker 78].

Our experience is that the major reason for "exponential blowup" of the search is the profuse application of axioms. One of the major reasons for this is probably that we do not consider axioms to increase the depth of the search for the purpose of pruning it. From studying traces of searches in our system, we believe that the caching mechanism is the single most important factor in limiting the otherwise profuse application of axioms.

We feel that the greatest shortcoming of our system is that the evaluation function has very little "understanding" of rotation, shifting, and bit extraction. More than two months were spent attempting to incorporate such knowledge into the system, but the effort was not successful. One of the reasons for our failure is that it appeared to us that it was necessary (at least logically) to have separate distance tables for each combination of rotations and bit lengths—an increase by a factor of 256 in the size of the distance tables for a 16-bit machine. We hope that this problem will be addressed more successfully in the future.

Chapter 7
Compaction

At the beginning of this research effort, our plan was to take the best microcode compaction algorithm available—which we believed to be that of Fisher [Fisher 79]—and to extend it to perform *interblock* compaction, particularly emphasizing the compaction of loops. As the research progressed, it became clear that there were still unsolved problems in the area of *intrablock* compaction; in particular, there are a large number of important code movements that current compaction algorithms do not consider. We also encountered problems in formalizing the interblock compaction constraints (see Section 2.2.2) because our micromachine model was more complex than that used by Fisher. As a result, we have limited our study to that of intrablock compaction.

We begin this chapter by reviewing Fisher's intrablock compaction algorithm, and then discuss two problems that his algorithm does not address; we believe that the second of these—the *data dependency problem*—is of fundamental importance. Finally, we present our compaction algorithm.

7.1. Fisher's Compaction Algorithm

The intrablock compaction algorithm of Fisher [Fisher 79], which compacts a linear sequence of μOps into μIs, consists of the following steps:

1. Determine the data dependencies among μOps based on register usage. A data dependency exists between two μOps *A* and *B*, where *A* precedes *B* in the original sequence, if *A* writes a register that *B* uses—ensuring that data is not read from a register before it is written—or if *A* reads or writes a register that *B* writes—ensuring that data in a register is not overwritten until all μOps that require its value have read it. The data dependencies in the latter group are actually *data antidependencies* [Banerjee 79]; as will be shown in Section 7.3, many important optimizations are missed because the algorithm treats them as data dependencies.
2. The height of each μOp in the dependency graph is computed.
3. The *data available set*—those μOps that have not been placed in a μI, but that are *data dependent* only on μOps that have already been placed in a μI—is computed.

4. The μOp from the *data available set* whose height is the largest among the μOps that do not conflict with the current μI is placed in into the current μI. If no such μOp exists, a new μI—which now becomes the current μI—is created, and the μOp from the *data available set* with the greatest height is placed into it.
5. Steps 3 and 4 are repeated until all μOps have been placed into μIs.

We are unable to use this algorithm without modifications for our machine model and compiler. One problem is that the algorithm assumes that the values of volatile registers do not extend across μI boundaries; another is that data dependencies are not handled in a general manner.

7.2. The Volatile Register Problem

The algorithm just described makes the assumption that at the end of every μI, the values of all volatile registers become undefined; thus data dependency constraints such as "μOp *A* must precede μOp *B* by exactly one μI" are not accounted for. Either two μOps must reside in the same μI—due to data being transmitted via a volatile register—or the second μOp may follow the first by an arbitrary number of μIs—in cases where data is transmitted via a non-volatile register. In the first case, the necessary simultaneous μOps are combined and treated as a single μOp (called a bundle) during compaction. In the second case, the μOps are treated as separate, but there is no upper bound on the distance between them; this guarantees that they can be compacted without backup.

When constraints are introduced that require two μOps to be a fixed distance apart, the notion of a *bundle* must be extended to include groups of μOps that do not all reside in the same μI, but whose placement relative to one another is fixed. The obvious extension of the algorithm is to map all data dependencies between μOps to data dependencies between bundles, and to map all conflicts between μOps to conflicts between bundles, taking care to account for the relative placement of any μOp within a bundle in all cases; the location of a bundle is defined to be the μI in which its earliest μOp(s) resides. Before such an extended bundle is assigned to a contiguous set of μIs, it is necessary to check conflicts with each μI. This extension, which was first proposed by Poe *et al.* [Poe 81], is the one that we use in our compaction algorithm.

During the latter stages of this research, we discovered a problem with this algorithm that arises because the presence of multi-μI bundles makes it possible for a bundle to be scheduled in an earlier μI than a bundle on which it is data dependent! Consider an example having the following constraints:

Bundle 1 — μOps *A*, *B*, *C* and *D* each belong to conflict class *X*, and must reside in consecutive μIs.

Bundle 2 — μOp *E* belongs to conflict class *Y*, and must not precede μOp *D*.

Bundle 3 — μOp *J* belongs to conflict class *X*, and must follow μOp *D*.

Bundle 4 — μOps *F*, *G* and *H* all belong to conflict class *Y*, and must reside in consecutive μls. In addition, μOp *H* may not precede μOp *J*.

The μOps are shown in Figure 7-1; the bracketed numbers along each dependency arc indicate the minimum and maximum relative placement between the two μOps. Figure 7-2 shows the same μOps, grouped into *bundles*. Notice that the minimum relative placement between bundles 3 and 4 is negative.

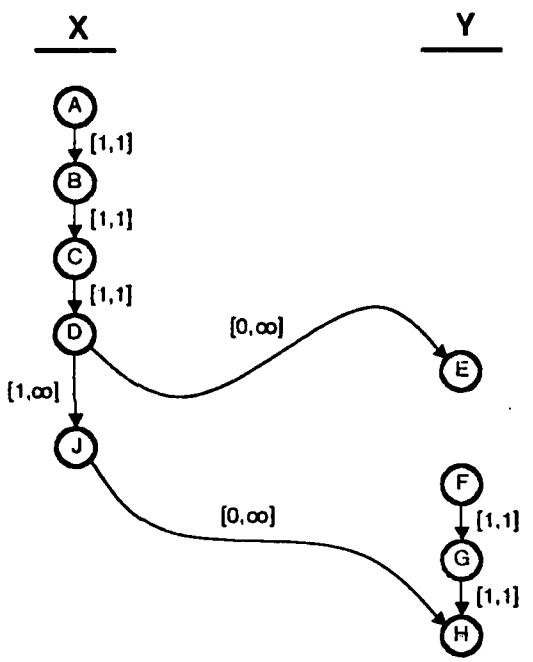

Figure 7-1: μOps with non-zero volatile data dependencies.

The proposed compaction algorithm would first place bundle 1 in μl 1; no bundles would be placed in μls 2 and 3 due to conflicts and data dependencies. When the algorithm reached μl 4, bundle 2 would be placed there. Bundles 3 and 4 could then be placed in μl 5. The resulting compaction, shown in Figure 7-3a, would have length 7.

Unfortunately, this compaction is non-optimal because the algorithm cannot anticipate the effect of a data dependency with a negative offset. A compaction of length 6 could have been obtained if the placement of bundle 2 had been delayed until after bundle 4 were placed in μl 3, as shown in Figure 7-3b. In order to obtain the optimal compaction, the algorithm must be modified to perform something similar to lookahead or backtracking.

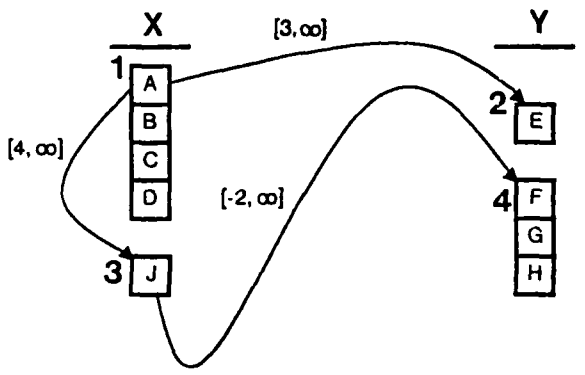

Figure 7-2: Bundles created from μOps in Figure 7-1.

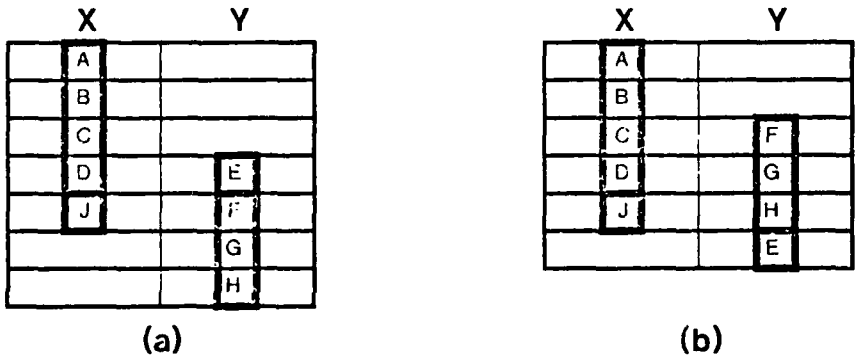

Figure 7-3: Compactions of bundles in Figure 7-2.

We therefore conclude that current *intrablock* compaction algorithms may do a poor job in the presence of constraints such as "must precede by exactly one", largely because it is sometimes necessary to consider μIs that are not *complete* (see 3.1.1.1) in order find the optimal solution, as was the case in the above example. Still, we are content to use the near-linear algorithm just described because we have devoted most of our research effort to other tasks. An extension of the *chain-matrix compaction algorithm*, presented in Section 7.3, can solve this problem in polynomial time, but the degree of the polynomial may be quite high.

7.3. The Data Dependency Problem

An even more serious problem than the one just discussed is that current compaction algorithms treat *data antidependencies* as *data dependencies*. Remember from Chapter 3 that a data antidependency is a constraint in which one μOp must precede another because the second destroys data that is read or written by the first.

Current compaction algorithms accept a linear sequence of μOps as input, and compute antidependencies solely on the basis of that linear order. This prevents such an algorithm from ever changing the order in which two μOps that write the same register are executed.

As an example of this problem, consider compacting the μOps

```
B <- A    (1)
X <- B    (2)
C <- X    (3)
X <- D    (4)
E <- X    (5)
F <- E    (6)
```

Data dependencies are placed among μOps *1*, *2* and *3*, and among μOps *4*, *5* and *6*; in addition an data antidependency is placed between μOps *3* and *4* because of their common use of register **X**, as is shown in Figure 7-4a.

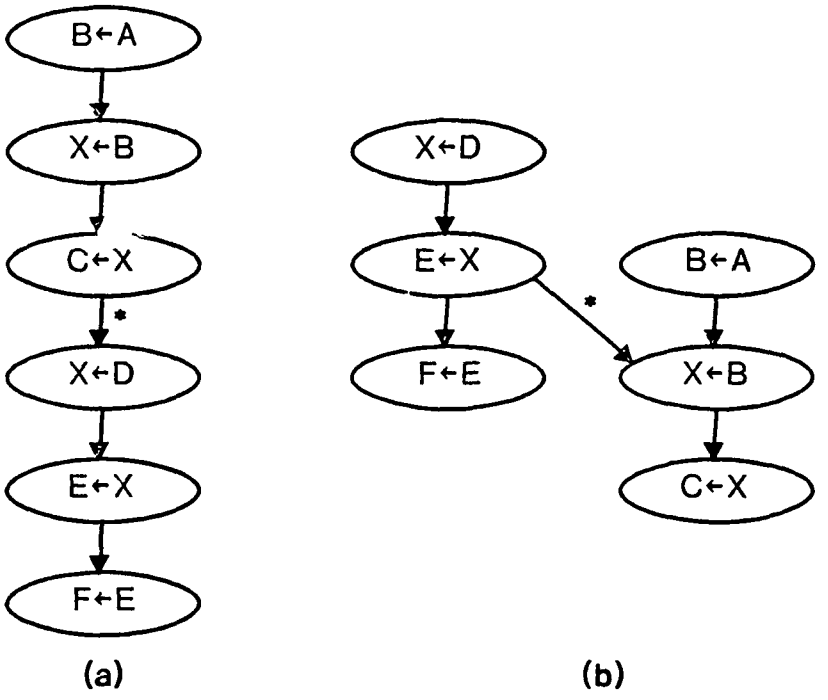

Figure 7-4: μOps with different data antidependencies.

This results in a μI sequence of length six because each μOp is data dependent—or antidependent—on its immediate predecessor. It is possible, however, to compact this sequence into four μIs if a different ordering is considered for the use of register **X**, as is seen in Figure 7-4b. Current compaction algorithms—even exhaustive searches—fail to consider such μOp movement.

We found this problem mentioned only once in the literature, and even then it was dismissed as unimportant [Fisher 81b]:

> As long as data precedence is not violated, a compacted microprogram will preserve its data integrity. A few integrity-preserving compactions that do violate precedence can sometimes be obtained by moving each write μOp and its associated reads as a group, but this is widely regarded as an excessively complicated technique offering little gain.

We believe this to be a misconception, which we suspect is due largely to the manner in which compaction algorithms have been tested. In some cases [Mallett 78], the test is based on an abstract machine model in which data dependencies and μl conflicts are assumed to be the only constraints; data antidependencies are not considered. In other cases [Fisher 79], μOps for a real micromachine are produced by taking hand-written code and uncompacting it; in this case the antidependencies—as determined by the original programmer—are (unwittingly) passed to the compaction algorithm.

In our compiler, μOps are passed to the compaction phase in the form of a dependency graph *in which data antidependencies—and hence the orderings for temporary registers—have not yet been determined*. We have no choice but to develop a method for determining the data antidependencies before—or in parallel with—compaction.

7.3.1. *Complexity revisited*

We now turn our attention to the problems of compacting microcode *with* and *without* predetermined data antidependencies. It is our contention that the problem of optimally ordering the μOps—and thereby determining the antidependencies—is the more difficult problem. We support this contention by proving (informally) that the compaction problem—once data antidependencies are specified—can be solved optimally in polynomial time. Because the general compaction problem is NP-hard, we conclude that the determination of antidependencies is likely the more difficult problem.

7.3.1.1. A polynomial-time algorithm

We base our proof on the commonly-accepted *classical microcode compaction* model [Fisher 79, Landskov 80]. The following properties are especially important:

- Two μOps that conflict may not be placed in the same μl.
- If one μOp is data dependent on another, the former may not precede the latter.
- The micromachine contains *v* registers, where *v* is a small constant. Two μOps that write the same register may not reside in the same μl.

Our proof depends particularly on the last item: the number of registers in the micromachine bounds the *breadth* of a data dependency graph to which antidependencies have been added. NP-hardness proofs of the compaction problem have assumed that the breadth of the graph could be arbitrarily large.

We now state the theorem, and sketch a proof.

> Theorem 1: An optimal solution to the *classical microcode compaction problem* can be discovered in time polynomial in the number of μOps, where the degree of the polynomial is equal to the number of registers in the micromachine.

We (informally) prove the theorem by sketching the *chain-matrix compaction algorithm*, which computes an optimal schedule in polynomial time. The overall strategy of the algorithm is to create a graph in the shape of a v-dimensional matrix, whose arcs represent legal μIs; the optimal solution is then determined by finding the shortest path in the matrix-graph from the origin to the node at the opposite corner.

This is accomplished by first dividing the μOps into v disjoint sets, called *chains*, according to the register each writes; if a μOp writes more than one register, it may be placed in the chain corresponding to either. According to the formulation of the problem, any two μOps that write the same register have, either directly or indirectly, a strict data dependency between them; thus, the data dependencies completely determine the order in which the elements of each chain are executed. The data dependency graph is therefore necessarily a set of v totally ordered *chains*, whose nodes may also have other data dependencies as well. An example of such a set of chains is shown in Figure 7-5. Data dependencies are represented in the figure by arcs (with the data dependencies belonging to chains are in bold face) and μOps are represented by nodes.

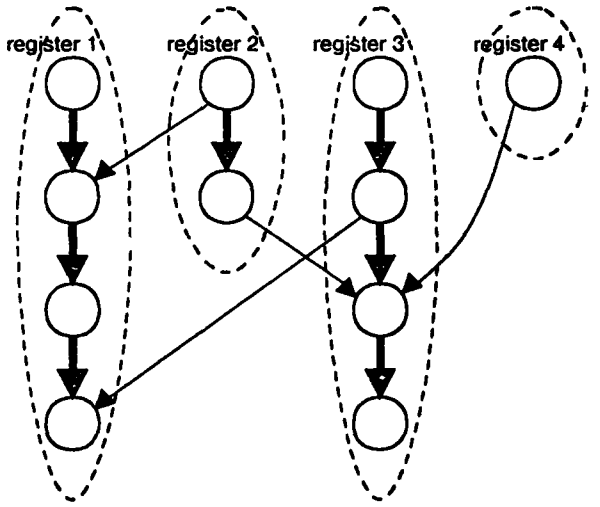

Figure 7-5: Data dependency graph cast as set of chains.

The compaction is performed by creating a graph in the shape of a v-dimensional matrix—one dimension for each chain—in which element $\langle k_1, k_2, \ldots, k_v \rangle$ of the matrix represents a partially completed sequence of μOps in which the first k_1 μOps from chain 1 have been

compacted, the first k_2 μOps from chain 2 have been compacted, and so forth. Directed arcs between elements of the graph represent μIs; the distance of an arc along any dimension must be either zero or one; an arc in the direction $\langle 1, 0, 0, \ldots, 0\rangle$ represents the μI containing only a μOp from chain 1, $\langle 1, 1, 0, 0, \ldots, 0\rangle$ represents the μI containing μOps from chain 1 and 2, and so forth. Each arc representing a μI that violates a conflict constraint is removed; likewise, each node of the matrix graph representing a set of μOps that violates a data dependency—that is to say, one that represents a situation where a μOp is compacted without one of its predecessors having also been compacted—is removed, along with any connected arcs.

At this point the problem is reduced to finding the shortest path from $\langle 0, 0, \ldots, 0\rangle$ to $\langle f_1, f_2, \ldots, f_v\rangle$, where the f_i are the lengths of the respective chains. Dynamic programming solutions to this problem are well known [Aho 74], and can be computed in time polynomial—in this case linear—in the number of nodes and arcs in the graph. If n is the total number of μOps, then the number of nodes in the matrix is certainly bounded by n^v, while the outgoing degree of any node is bounded by a constant—namely 2^v. Thus the complexity of the algorithm is $O(n^v)$, where v is the number of registers in the micromachine.

7.3.1.2. An example

As an example, consider the μOps in Figure 7-6.[7] The solid lines represent data dependencies, while the dotted lines represent conflicts. The data dependency arc marked with an " = " denotes a non-strict dependency—that is, a dependency in which the μOps are allowed to reside in the same μI. Strict dependencies are "implemented" by a non-strict dependency and a conflict. The bold lines represent strict dependencies between elements of a chain. The matrix-graph for this problem, shown in Figure 7-7, has been augmented with markings that illustrate the mapping from the original problem. The node marked **X** represents a μI sequence into which μOps *A*, *B* and *E* have been compacted, the arc marked **Y** represents the μI containing μOps *C* and *F*, and the arcs marked **Z** each represent the μI containing only μOp *G*. In order to include conflict and data dependency information in the matrix-graph, arcs representing illegal μIs, and nodes representing sets of μOps that violate data dependency, are removed. This means that 7 arcs,

(A,E) (B,E) (B,F) (C,E) (C,G) (C,H) *and* **(D,G)**

and 8 nodes are deleted. The node in the bottom-left corner, for example, is removed because it represents μIs containing μOps *A*, *B*, *C*, and *D*, which violates the constraints that *C* must not precede *E* and that *D* must not precede *G*. Figure 7-8 shows the modified

[7] We use an example with only two chains because a matrix of dimension two is much easier to depict on paper than one of higher dimension.

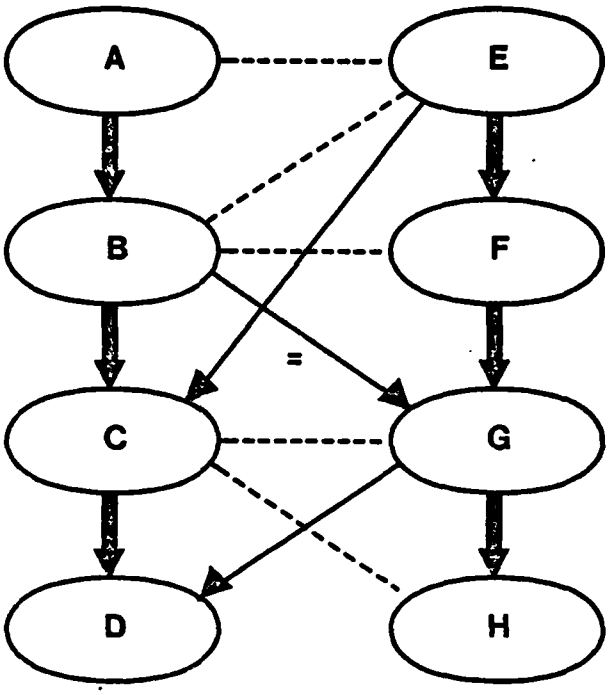

Figure 7-6: Data dependency graph with conflicts.

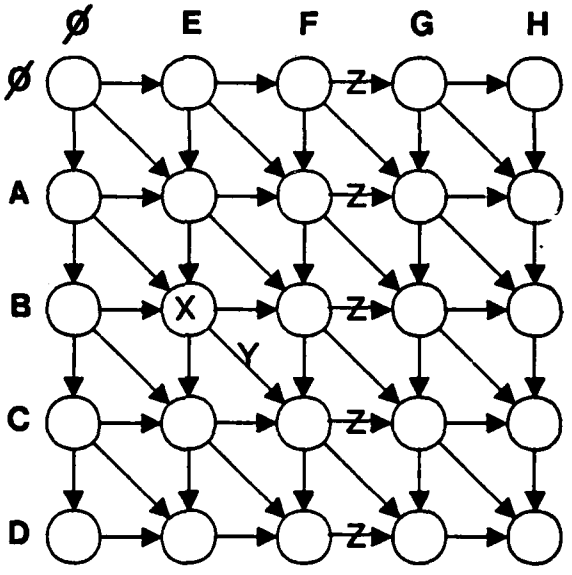

Figure 7-7: Matrix-graph before modifications for constraints.

matrix-graph, in which each node is also marked with its distance from the origin. A minimal path—shown in bold face—is produced by following arcs from the final state (i.e., bottom-

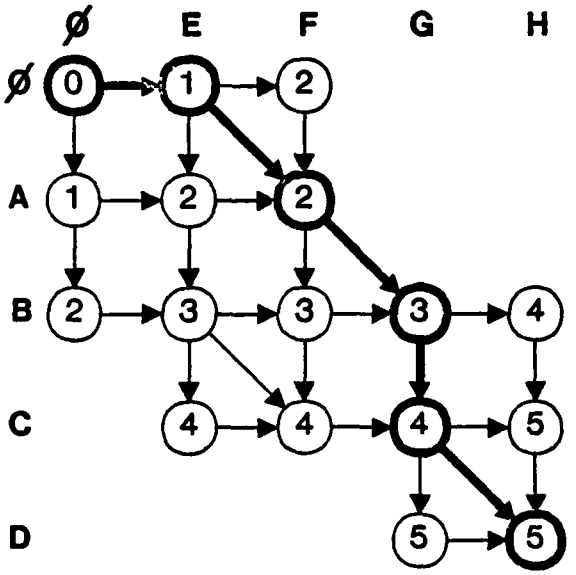

Figure 7-8: Matrix-graph after modifications for constraints.

right corner) that always reduce the distance by one. The resulting μI sequence is shown in Figure 7-9.

We conclude from Theorem 1 that although the local microcode compaction problem is NP-hard, the addition of a "complete" set of antidependency arcs to the dependency graph constrains the breadth of the graph so severely that problem can be solved in polynomial time. A corollary is that the determination of the initial ordering of μOps for the purpose of determining antidependencies is NP-hard and is therefore likely a more difficult problem than that of compaction with predetermined antidependencies.

7.3.1.3. Main memory references

Before proceeding any further, we wish to address the question of references to an arbitrarily large external memory. If each memory location is considered to be a *register*, the algorithm is again exponential. We answer this by observing that writes to external memory are typically performed on micromachines by first loading the data and memory address into "memory data" and "memory address" registers, and then performing the actual transfer. The order in which writes are made to main memory is thus completely determined by the order of the μOps that write the micromachine registers that hold the data and address; this allows the external memory to be treated as a single register. The result does not apply to machines in which data may be written to main memory without first being loaded into a register.

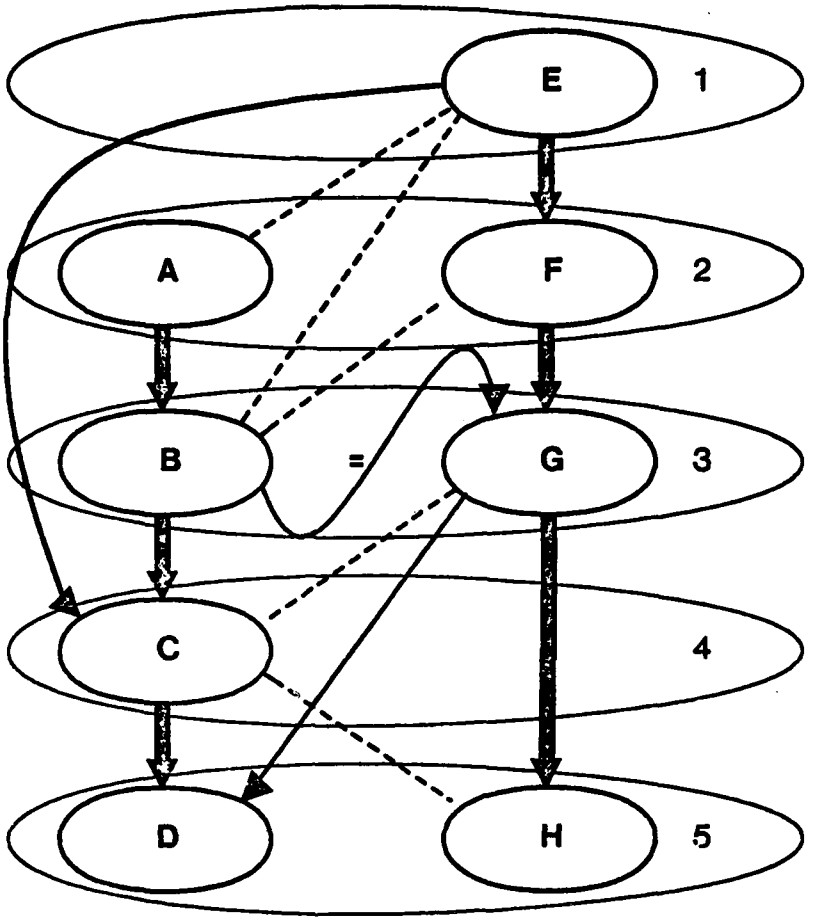

Figure 7-9: Optimally-compacted μOps.

7.3.1.4. More complex machine models

The chain-matrix algorithm—and hence the complexity result—can also be applied to slightly more complex micromachine models, two of which we will mention here. The first extension allows the data dependency discussed in Section 7.2, "μOp *A* must precede μOp *B* by exactly one cycle," to be expressed. Because a node in the matrix-graph represents a set of μOps, the restriction "may follow by no more than one μI" can be enforced by removing all arcs that originate from nodes (μIs) "containing" μOp *A*, but whose destinations do not "contain" μOp *B*. The addition of a strict data dependency between the μOps can be used to guarantee that μOp *B* follows μOp *A*. Together, the two restrictions satisfy the original constraint. A dependency of the form "μOp *A* must coincide with μOp *B*, or precede it by exactly one cycle" may be modeled in an analogous manner, using a non-strict dependency.

The algorithm can also be extended to micromachines that allow some registers to be written twice during the same microcycle. This is done by allowing the arcs in the matrix-graph to have a length of two along dimensions that correspond to those registers.

7.3.2. Our solution

Because our compaction algorithm begins with a data dependency graph of μOps instead of a linear sequence of μOps, we must ensure that overlapping uses of a register do not occur. We have considered two methods of performing this task. The first was to develop a completely new compaction algorithm that accounts for register conflicts as it compacts the μOps. Although we suspect that this approach will ultimately lead to the best solutions, we reject it for our system because such an algorithm would almost certainly entail heuristic search and backtracking; its development alone would require a substantial research effort. Instead, we adopted a second approach: that of pre-serializing the graph using a *simple* heuristic, thereby making it amenable to a compaction algorithm in which the antidependencies are assumed to be specified.

In general, a dependency graph specifies only a partial ordering, while the equivalent of a total ordering is needed to compute a *complete* set of antidependencies. Our approach is to give commonly-used registers the highest priority, allowing infrequently-used registers to be hold their values for longer periods of time. For the purpose of defining priority, we consider *volatile* registers to be used "infinitely often", thereby guaranteeing that each use of such a register is localized. Thus the serialization algorithm ranks all registers—first according to volatility, and then according to frequency of use—and then ranks the μOps by iteratively binding dependent μOps together in order of the priority of their dependency.

As an example, consider the data dependency graph in Figure 7-10, where the nodes represent μOps, and where each dependency (arc) is marked with its "ranking"; the dummy μOps X_0 and X_F have been added to indicate registers that are *live* at the beginning or end of the sequence.

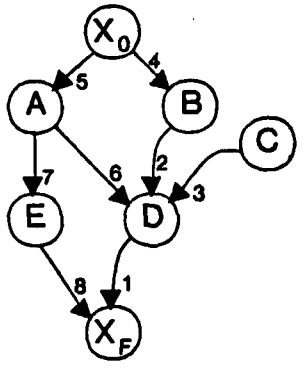

Figure 7-10: Dependency graph before serialization.

First, *D* is placed before X_F and *B* is placed before *D* because the dependencies between those pairs are of the highest priority.

X_0 ... B D X_F

Then *C*, based on its dependency with *D*, is placed next to *B*, which is the closest available position before *D*.

X_0 ... C B D X_F

The next dependency, which is between X_0 and *B*, is ignored because *B* has already been placed. The remaining μOps, *A* and *E* are placed between X_0 and *C*.

X_0 A E C B D X_F

Because this algorithm does no backtracking, it is possible for an illegal serialization—one in which a register is required to hold two distinct values simultaneously—to be produced. One reason is that it is possible for a code generator to produce code for which no legal serialization exists (Figure 7-11a); we have produced (by hand) cases where the algorithm would even fail to find a serialization that exists (Figure 7-11b). The algorithm checks for such inconsistencies, but gives only a warning if one occurs. We do not examine the problem further in this dissertation because such a situation has never occurred during our experiments, and because we suspect that "higher-level" issues, such as register allocation, are also involved.

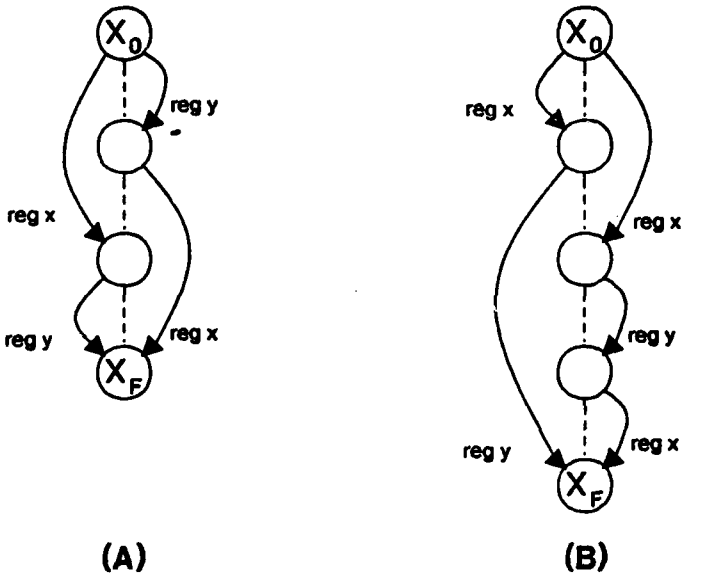

Figure 7-11: Illegal serial orderings of μOps.

7.4. The Intrablock Compaction Algorithm

We now present the algorithm that compacts a dependency graph of μOps into μIs.

1. First, the serialization algorithm described in Section 7.3.2 is used to place the μOps into a linear sequence that satisfies the data dependency constraints.
2. Then antidependencies are placed between any two μOps in which the first precedes the second in the linear list and reads or writes a register that the second writes. From this point on, data dependencies and antidependencies are treated identically.
3. Next, the μOps are mapped into bundles. Any two μOps that are required, according to the data dependency graph, to reside a constant distance apart are placed into the same bundle. Data dependencies between μOps in different bundles are mapped into data dependencies between their respective bundles in a way that accounts for the relative position of each μOp within its bundle. Conflicts from all μOps in a bundle are mapped into the bundle's conflict list; again the relative position of the each μOp is taken into account.
4. Finally, the height of each bundle in the data dependency graph is computed in the obvious way, and the compaction algorithm of 3.1.1.2—modified to handle multi-μI bundles as described in Section 7.2—is applied, where bundle height is used as the evaluation function, with the highest bundles placed first.

7.5. Summary

The major result of this chapter is not a new compaction algorithm, but rather a demonstration that previous intrablock compaction algorithms are inadequate because they rely on the order in which the μOps are placed in the source code to determine the placement of data antidependencies. We have shown that the complexity of the problem solved by such algorithms is polynomial in the number of μOps, and have therefore concluded that the difficult part of the compaction problem is the initial placement of the data antidependencies. *We therefore do not consider intrablock compaction to be a solved problem, as seems to be the general consensus among researchers in the field* [Davidson 81].

We have also presented a modest extension to the intrablock compaction algorithm of Fisher that addresses the data dependency problem and handles volatile registers in a more general—but still inadequate—manner. It certainly will not always produce optimal code, but it has performed well in our limited experiments.

Chapter 8
Coupling Code Generation and Compaction

This chapter describes the methods by which we attempted to couple the code generation and compaction phases of the compiler. Each method succeeded—that is, produced better code than without coupling—in some situations, but failed in others; sometimes the coupling perturbed the search so severely that no code was produced at all.

Recall from Chapter 4 that three methods of coupling were tested. The first requires the compaction phase to select one of several of code sequences that have been produced by the code generator. The second involves a feedback loop between the two phases, while the third requires the code generator to "call" the compaction phase as a subroutine, using information returned to prune the heuristic search.

The first section describes three example problems—along with their solutions—used in this chapter to illustrate the strengths and weaknesses of each method. The next three sections describe the coupling methods, reporting their behavior on the three "test problems", and give summaries of their effectiveness. Finally, an attempt to combine two of the coupling methods is described.

8.1. Illustrative Problems

The example problems used in this chapter are all from the Kmap [Ousterhout 78] microarchitecture. (A sketch of the Kmap may be found in Appendix D.) Due to the length of the heuristic searches described in this section, it is not feasible to present them in the text. Traces of some, however, along with examples from the Puma [Grishman 78] microarchitecture, may be found in Appendix F.

So that the reader may better understand the examples in this chapter, we first discuss relevant features of the Kmap. The two ALU data inputs are *areg* and *breg*; there do not exist, however, ALU functions "select *areg*" or "select *breg*". The "normal" way to move the *areg* value to the *fbus* (i.e., ALU output) is to put the constant "-1" in *breg*, using the *breg.ones* μOp, and to set the ALU function to *AND*. Similarly, the value of *breg* may be passed to the *fbus* by placing the constant "0" in *areg* and setting the *ALU* function to *OR*. This is more

difficult for the heuristic search to discover, however, because there is no μOp that explicitly sets *areg* to zero. Instead, the μOp

```
(<- areg (and %mask (rot scount tlatch))
```

is used, requiring the search to apply the axiom "zero *AND*ed with anything is zero" and to recognize that `%mask` pattern matches "0".

The first problem we will consider is that of producing the constant "−2" on the *fbus*. It was chosen largely because it was the only example in which the *squeeze* method outperformed the others. We find it an interesting task, because the optimal solution is quite difficult to discover.

The first sequence (see Figure 8-1) takes advantage of the fact that the *constant register* is directly connected to *breg*. The *constant register* is loaded, and the value is then moved to *breg*. A zero is placed in *areg* and an *OR* ALU function causes the value to appear the *fbus*.

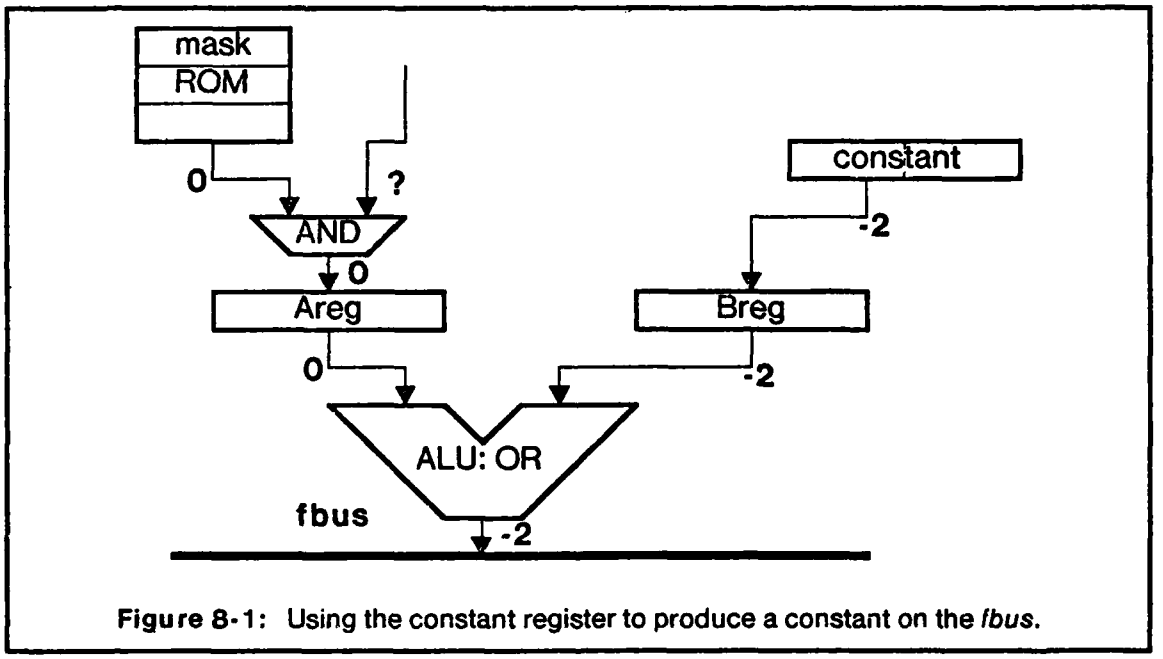

Figure 8-1: Using the constant register to produce a constant on the *fbus*.

The use of the constant in the Kmap tends to make the literal field a bottleneck because two μOps—that both use the literal field—are required to load the constant—one for the high half, and one for the low half.

Another sequence, which is the one generated without coupling, produces the constant in *areg* using the "−2" mask (see Figure 8-2). This requires a "−1" to be placed in *tlatch*, having been routed from the *fbus* via the *abus*. We remark that this sequence requires the *fbus* to be used during two μIs—one with the ALU function *ONES*, the other with the function *OR*.

The best method for producing the constant, however, is to perform a subtraction using the

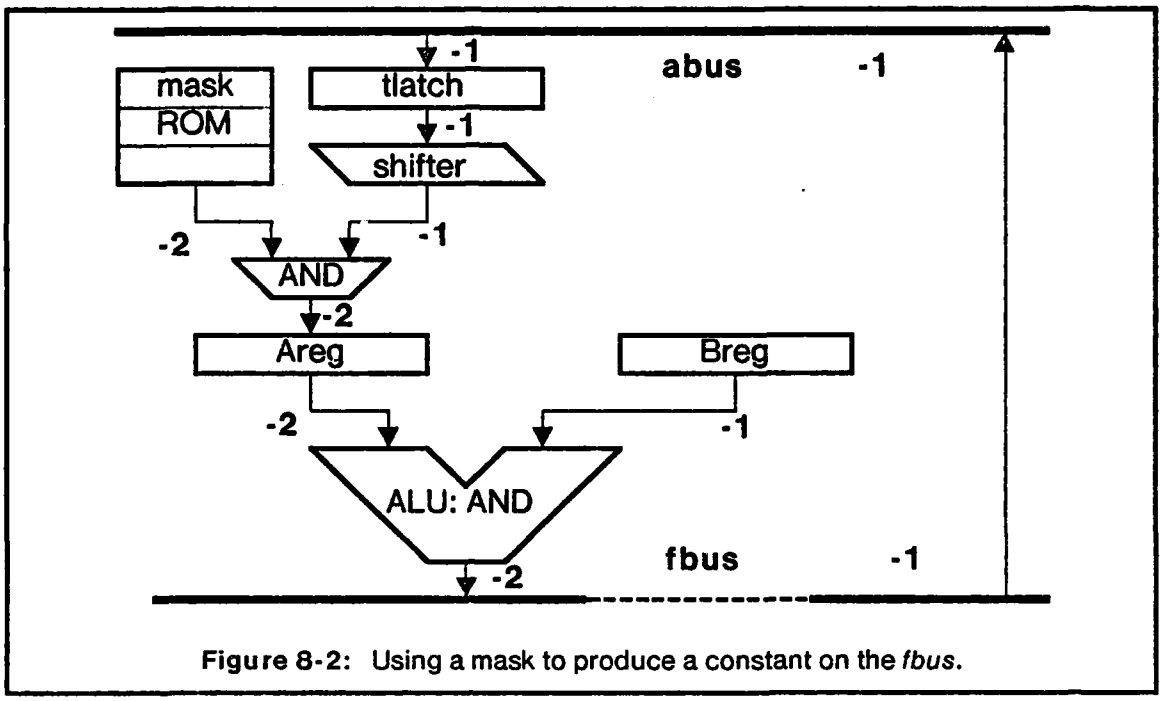

Figure 8-2: Using a mask to produce a constant on the *fbus*.

μOp (`<- fbus (+ (+ (not areg) breg) carryin)`). When the carryin is set to zero, this μOp is effectively *breg* – *areg* – 1. Because *breg* can be easily set to "–1", and *areg* to "0", a "–2" can be produced on the *fbus* in this way without using any resource for more than one cycle. Unfortunately, discovering this sequence requires a number of axioms to be applied; specifically, the constant "–2" must be unfolded, through the repetitive application of axioms and selection of μOps into

```
(+ (+ (not (and 0 (rot scount tlatch))) -1) 0)
```

which is quite difficult to discover.

The second and third examples require code to be generated that places the constant "7" on the *fbus*, while performing an additional task. In the second example the additional task is that of moving data from *lincwd* to a word in the *dram* (data ram); in the third example, a word must be copied from the *dram* to a *gpr* (general purpose register).

There are two basic ways to produce the constant "7" in the Kmap; the first, which uses the constant register, has the drawback that it requires the literal field to be used during two μIs while both halves of the constant register are loaded. This can produce poor code if other operations that use the literal field—such as loading *dram* or reading *lincwd*—are nearby, because the literal field will be a bottleneck. The second method of generating a "7" is to use the mask unit, as "7" is one of the available masks. As in the first example, this requires the production of a "–1" from the *fbus*; thus, the method "overloads" the *fbus*.

In both cases, the code sequence using the mask was produced by the code generator in

the absence of coupling. We would expect that *with* coupling, the constant register would be used in the latter case, as the literal field is otherwise free.

8.2. And/Or Method

The *And/Or* method of coupling the two phases requires the code generator to produce several code sequences so that the compaction phase can select the one that produces the shortest μI sequence. This is implemented by modifying the *search* and *transform* routines to return an *And/Or tree* [Winston 77] of μOps. Recall from Chapter 4 that an *And/Or tree* is a tree in which each interior node is marked either *And* or *Or*; a solution to a tree whose root is marked *And* consists of solutions for all of its sons, while a solution to a tree whose root is marked *Or* consists of a solution for any of its sons. Because an *And/Or tree* represents a set of solutions, it is the responsibility of the compaction routine to choose the solution that produces the smallest final code.

8.2.1. Modifications to the code generation and compaction routines

Recall that the code generation algorithm in Chapter 6 produces a degenerate *And/Or tree*—in which all interior nodes are *And* nodes—representing only a single solution consisting of all μOps in the tree. The *And/Or* coupling method considers trees in which *Or* nodes are also present; this requires both that the code generator be modified so that it produces such trees, and that the compaction routine be modified so that it accepts them as input.

Enabling the code generator to produce multiple solutions is reasonably straightforward. The *search* and *transform* routines are modified so that each continues searching even after a solution is found; two or more solutions for a given subproblem are placed under an *Or* node in the *And/Or tree*. Thus, each recursive call to *search* or *transform* has the potential to produce an *Or* node in the tree.

The modification to the compaction routine is more difficult. Although in theory a compaction could be attempted for every combination of μOps in the set of solutions specified by the *And/Or tree*, the number of solutions grows exponentially with the depth of the tree; such an approach is therefore acceptable only for small trees. We have adopted a hill-climbing strategy [Winston 77] that considers each leaf node at least once, but does not consider all combinations of μOps.

Initially, the cheapest sequence of μOps, according to the μOp cost table, is selected; we will call this sequence the *primary sequence*. Then a set of *secondary sequences* are computed from the primary sequence. A secondary sequence is a group of μOps that differs from the primary sequence "in only a few μOps". More precisely, a group of μOps is a

secondary sequence if it can be transformed into the primary sequence by changing the selection of exactly one *Or* node in the *And/Or tree*. The primary sequence and each of the secondary sequences are compacted. The sequence that compacts most tightly is chosen as the new primary sequence, and the process is repeated until no secondary sequence can be found that is better than the current primary sequence. Ties are broken by first comparing the number of subcycles used by each sequence and then the total cost of the μOps as defined by the μOp cost tables.

As a simple example, let us name the μOps *m1* through *m6*, and assume that the *And/Or tree*, shown in Figure 8-3, is ordered so that the left-most operands are the ones considered least expensive.

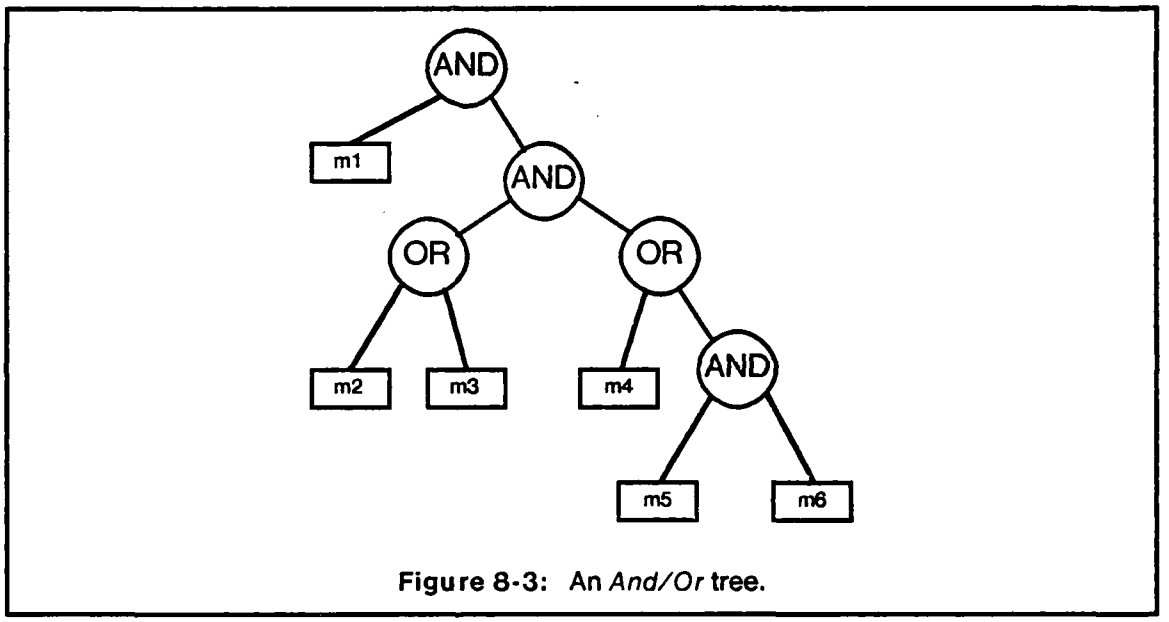

Figure 8-3: An *And/Or* tree.

The primary sequence is

m1 m2 m4

where all *right* sons of *OR* nodes are pruned away. The two secondary sequences,

m1 m3 m4 *and* **m1 m2 m5 m6**

are computed by reversing the sense of the first and second *OR* nodes respectively. Let us assume that the sequence

m1 m2 m5 m6

compacts most tightly. Then it becomes the new primary sequence, and the secondary sequences are

m1 m2 m4 *and* **m1 m3 m5 m6**

In practice, there would be more than two *OR* nodes, and this process might continue for several iterations.

8.2.2. Examples

In the first example,

```
(<- fbus -2)
```

the search examines 64 nodes and finds the following code sequences:

1. Loading the constant register with "-2", gating it onto *breg*, masking *areg* with zero, and performing an *OR* operation in the ALU.
2. Moving a "-1" from the *fbus* to *tlatch*, masking it with a "-2" into *areg*, putting "-1" on *breg* and performing an *AND* operation in the ALU.
3. Same as (2), except that a *gpr* is allocated and used to pass the "-1" from the *fbus* to *tlatch*.
4. Same as (2), except that a zero is placed in *breg* (via the *fbus* and *fblatch*), and an *OR* is performed in the ALU.

The first sequence is initially chosen as the primary sequence, but after all compactions are attempted, it is discovered that the second requires one less μI. A second iteration with (2) as the primary sequence uncovers no new sequences, so (2) is selected as the best sequence. Without more powerful heuristics in the code generator, the optimal (subtraction) sequence was not found.

In addition, the *And/Or* method did not discover the sequence using the constant register until after we "precompiled" the solution to

```
(<- areg 0)
```

This same precompilation was also necessary for the other *And/Or* examples described in this section.

In the second example, the source statements

```
(; (<- dram[dadr 0] lincwd) (<- fbus 7))
```

are compiled and compacted. In the Kmap, both accessing *lincwd* and writing the *dram* require the use of the literal field of the μI; one would thus expect a poor compaction from a sequence that generates the "7" by loading it into the constant register because it uses the literal field for two cycles. On the other hand, loading a *gpr* from the *dram* does not require the literal field to be used.

Only one sequence is found to move data from *lincwd* to the *dram*, but five are produced to put the constant "7" on the *fbus*:

1. Use the constant register to generate the "7", setting *areg* to "0", and setting the ALU function to *OR*.
2. Use the mask to generate the "7", fetching a "-1" from the fbus, as was done in the previous example, and setting *breg* to "-1" and the ALU function to *AND*.
3. Same as (2), but using a *gpr* to store the "-1" for one or more cycles.

4. Same as (2), but putting a "0" in *breg* (via *fbus* and *fblatch*), and setting the ALU function to *OR*.

5. Same as (4), but using a *gpr* to store the "-1" for one or more cycles.

Initially, the sequence that uses the constant register is chosen as the primary sequence, but is replaced on the next iteration because it requires 5 cycles to compact, while all of the others require only three; this is, of course, due to the heavy use of the literal field. Sequence (2) is finally chosen as the best sequence.

The third example,

```
(; (<- gpr[2] dram[dadr 0]) (<- fbus 7))
```

is a different matter. In this case the first source statement does not use the literal field of the μl, but rather uses the *fbus* for an additional cycle; thus, the constant register is used to generate the "7".

8.2.3. Evaluation

The *And/Or* method of coupling the phases appears to be an effective one. Once the *And/Or* tree has been generated, the compaction phase seems to have little trouble selecting a good sequence. In particular this method has performed well in situations similar to that described in 4.1.1.3. We remark, however, that all of our experiments have been moderately small (e.g., 100-200 nodes); we would not necessarily expect the hill-climbing to perform as well with a larger tree—say several thousand nodes—as input.

The major difficulties appear to be in controlling and directing the code-generation process. One problem we have encountered has been excessive searching even after acceptable solutions have been found. Because the evaluation function often overestimates the difficulty of producing code for a particular expression, it is possible for the *search* and *transform* routines to "waste" a large amount of time attempting to find additional solutions that may not even exist. In order to contain the search, we have introduced a global search parameter that we call the *foundfactor*, which is typically a real number in the range (0, 1). Whenever a code sequence is found that satisfies a particular invocation of *search* or *transform*, the cutoff is multiplied by the *foundfactor*; additional solutions are thus required to satisfy a more stringent cutoff. If a second solution is found, the cutoff is again multiplied by the foundfactor, further limiting the depth of a search for a third sequence.

We have generally set the initial search cutoff to be 1.2 times its estimated cost as determined by the evaluation function. A foundfactor of 0.84—so the product of the two is slightly greater than 1.0—has generally performed well in our experiments. This tends to allow at least two solutions to be found at any given level of the search. Figure 8-4 illustrates the effect of the foundfactor on the search.

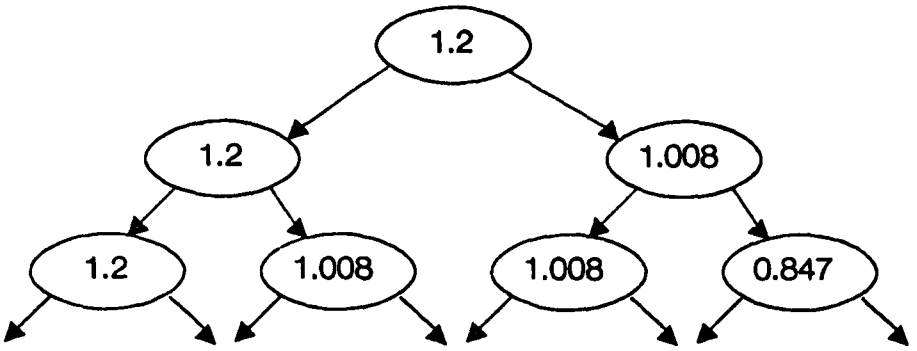

Figure 8-4: Illustration of cutoff being reduced with search breadth.

One of the shortcomings of *And/Or* method is that occasionally a simple solution is found, but the search continues, attempting to find more complicated solutions. This was made painfully clear during an experiment in which a *search* for the subgoal,

```
(<- fbus 0)
```

was passed a relatively large cutoff. In the Kmap microarchitecture, there is an explicit μOp that performs the function of setting the *fbus* to zero. The large cutoff, however, allowed the search to continue to find "better" solutions, such as

```
(<- fbus (and (and 0 (rot scount tlatch)) breg))
    and
(<- fbus (+ (+ (and 0 (rot scount tlatch)) (not -1)) 0))
```

Cattell addressed this problem by introducing a *breadth limit*; when the number of nodes traversed in the search tree during a search for an additional solution exceeded a predefined limit, it was terminated. We have had difficulty directly applying his solution to our system, because the breadth limit was defined to be a function of the depth; in our system, there is little correlation between the absolute depth of the search and the amount of work to which we are willing to expend in finding a solution.

Another problem we encountered with the *And/Or* method was that of finding redundant solutions, which can happen when the order in which axioms are applied is reversed. Figures 8-5 and 8-6, for example, show two nondeterministic searches that find the same solution to a problem. In many cases, the order in which the μOps are generated is different, so such redundancy does not become apparent until the search is completed. Such redundant solutions may cause the cutoff to be reduced to the point that other unique solutions are missed. Although certain features of the searching strategy—requiring destination operands to match when considering a feasible μOp, for example—reduce the number of duplications, it is not uncommon for our system to discover the same sequence of μOps in four or five different ways.

```
transform: (+ 0 breg) => (+ (+ areg breg) carryin)
    apply commutativity axiom
 transform: (+ breg 0) => (+ (+ areg breg) carryin)
     decompose by operand
  transform: 0 => carryin
      results in μOp: carry.0
  transform: breg => (+ areg breg)
      apply identity axiom
   transform: (+ 0 breg) => (+ areg breg)
       decompose by operand
    transform: 0 => areg
        results in μOp: areg.mask 0
```

Figure 8-5: Example of transform function.

```
transform: (+ 0 breg) => (+ (+ areg breg) carryin)
    apply identity axiom
 transform: (+ 0 (+ 0 breg)) => (+ (+ areg breg) carryin)
     apply commutativity axiom
  transform: (+ (+ 0 breg) 0) => (+ (+ areg breg) carryin)
      decompose by operand
   transform: 0 => carryin
       results in μOp: carry.0
   transform: (+ 0 breg) => (+ areg breg)
       decompose by operand
    transform: 0 => areg
        results in μOp: areg.mask 0
```

Figure 8-6: Redundant version of transform in Figure 8-5.

An inherent problem with the *And/Or* strategy is that the code generator receives no feedback from the compaction phase; it must therefore be "intelligent" enough to create all possible code sequences that might compact well in a given situation. In the first example, the code generator in fact did *not* find the best solution, because it required the application of more axioms than did other solutions. We see this as the most fundamental problem; if the code generator is good enough—a big *if*—we believe that the *And/Or* method can be used to produce high-quality compacted microcode.

8.3. Iteration

The *iteration* method requires neither the code generation nor compaction phases to be modified; rather, a post-compaction analysis is performed on the compacted microcode to determine which μOps are responsible for causing bottlenecks. The cost tables, which are used by the code generator to guide the search, are then modified so that "bottleneck-prone" μOps are assigned a higher cost, and the search is repeated. The idea is to encourage the code generator to use μOps that are less likely to conflict with other μOps.

8.3.1. Post-compaction analysis

The post-compaction analysis consists of two phases. The first is the determination of which conflicts are most often involved in bottlenecks. The second is the updating of the conflict cost tables, which are in turn reflected in the μOp cost tables and distance tables.

Our first attempt at post-compaction analysis was to count the number of times that a conflict was present in the μOps produced by the code generator, and increase the cost of the conflict(s) that appeared the most frequently. This strategy had the drawback, however, that conflicts appearing frequently were penalized, rather than ones that might have caused local bottlenecks.

This led us to change our approach: instead of counting the conflicts, the μOps are first divided into bundles—a set of μOps that is compacted as a group (see 2.2.5.2). Then, one of the bundles is removed, and the remaining bundles are compacted; if this "modified" microcode compacts more tightly, we assume that the removed bundle must have contained a bottleneck. Following this, the bundle is returned to its original place, and another bundle is chosen for removal; this process is performed for each bundle. Each conflict contained in any "bottleneck-prone" bundle becomes a candidate for having its cost increased.

In determining the quantity to add to each conflict, we have taken the approach that the sum of the conflicts' costs should increase by constant amount—in our experiments 10 units—during each iteration; there is therefore a finite amount of "cost" to be allocated among conflicts that are involved in bottlenecks. This cost is allocated in proportion to the product of the *conflict's current cost* and *total number of μIs "saved"* during compactions in which a bundle containing the conflict was "missing". In the current implementation, these costs are represented by integers, so the computations are only approximate.

As an example, let us assume that conflicts involving

`alu`	*(cost 5)*
`regfile`	*(cost 3)*
`shifter`	*(cost 6)*
`literal`	*(cost 8)*

exist, and that three bundles have been produced by the code generator, containing the conflicts

```
[alu literal]
[shifter]
  and
[alu regfile]
```

respectively. Let us further assume that when the code is compacted without the `[alu literal]` bundle, two μIs were saved, that none were saved when the `[shifter]` bundle was removed, and that one was saved when the `[alu regfile]` bundle was

removed. If we desire to add a cost of 10 to the set of conflicts, the *alu* conflict is increased by 4, and *literal* conflict by 5, and the *regfile* conflict by 1; these increments are computed as follows:

conflict	*orig. cost*	*μIs saved*	*product*	*proportion*	*×10, rounded*
alu	5	3	15	0.44	4
regfile	3	1	3	0.09	1
shifter	6	0	0	0.00	0
literal	8	2	16	0.47	5

In addition to the modification of conflict, μOp, and distance tables, the caches must also be flushed, so that information based on the old table values is not present.

8.3.2. Examples

The examples illustrate the reasons that we found this coupling method rather disappointing. Before we present the examples, however, we wish to define some terminology so that two different types of iteration are not confused. When a search is initiated, it is passed a cutoff that is computed by multiplying its "expected cost" (as determined by the evaluation function) by a small factor such as 1.2. If the search terminates in a failure, this factor is increased and the search is attempted again. We shall call these failure-induced repetitions *subiterations*.

At a higher level, we speak of *iteration* to mean the cycle in which code is generated, code is compacted, tables are updated, code is generated, and so forth. The purpose of this iteration is to improve code that has already been successfully generated; we call these improvement-induced repetitions *iterations*. Therefore the statement "the first iteration required only one subiteration, but the second required three," means that the search using unmodified tables was successful the first time, but that it took three searches (with successively greater cutoffs) in order to find a code sequence after the tables were modified.

In the first example, where the constant "-2" is to be placed on the *fbus*, the algorithm found a 2-μI solution—using the mask unit—on the first two iterations, and then found a 3-μI solution—using the constant register—on the next two iterations. On the fifth iteration, no solution was found after the first two subiterations, and the third gave indications of taking a *very long time*, at which point we manually terminated the search. Table 8-1 summarizes its performance on the first example. The distressing result is that as the tables become "better", the cost of finding a solution increases, and the quality of the solution decreases.

In the second example (see Table 8-2), where it is undesirable to use the constant register because of literal field conflicts, a 3-μI sequence is generated on the first iteration. On the second iteration, the algorithm perceives the *fbus* as a bottleneck, and the task of constant generation is assigned to the constant register, resulting in a 5-μI sequence. On the third

iteration	*subiterations*	*total # nodes*	*# µIs*	*comments*
(1)	1	23	2	uses mask
(2)	2	34	2	same as (1)
(3)	2	41	3	uses const. reg.
(4)	3	61	3	same as (3)
(5)	2	333	??	no solution after 333 nodes

Table 8-1: Summary of first iteration coupling example.

iteration, the sequence using the constant register is again found, but at greater search cost. Finally, the solution using the mask unit and *fbus* is rediscovered on the fourth iteration.

iteration	*subiterations*	*total # nodes*	*# µIs*	*comments*
(1)	1	34	3	uses mask
(2)	2	57	5	uses constant register
(3)	3	68	5	same as (2)
(4)	4	114	3	same as (1)

Table 8-2: Summary of second iteration coupling example.

In the third example (see Table 8-3), the goal of putting a "7" on the *fbus* should be achieved using the constant register, as the literal field is otherwise unused. In this case, as in the previous examples, the solution using the mask is generated on the first iteration; in this example, however, an identical search is performed during the second iteration. Finally the solution using the constant register is found on the third (and again on the fourth) iteration, decreasing code size from 4 to 3 µIs.

iteration	*subiterations*	*total # nodes*	*# µIs*	*comments*
(1)	1	49	4	uses mask
(2)	1	49	4	same as (1)
(3)	2	73	3	uses constant register
(4)	2	73	3	same as (3)

Table 8-3: Summary of third iteration coupling example.

8.3.3. Evaluation

We found these results rather discouraging, as we had hoped for a quick convergence to a good solution in most cases. More than one code sequence was found for each input

expression, but the convergence to good solutions was not impressive. Furthermore, the amount of time spent finding a solution tended to increase with each iteration; one would have hoped that the finding a solution would become easier as the cost tables became "better".

We have two theories for the reason that the cost increases with each iteration. The first is that when the cost of some μOps is increased, the initial estimate of the cost of the search—and hence its depth—is also increased. Thus, a search is allowed to go deeper if it involves only μOps whose costs did not increase; in many cases such searches are fruitless anyway. The other theory is that there are many times when it is impossible to generate code that completely avoids using a "high-cost" conflict, so the goal becomes one of minimizing its use; if a particular conflict is assigned an extremely high cost, the distinction between the costs of other conflicts can become "noise", causing the evaluation function to become less effective.

Another shortcoming of the *iteration coupling* method is that it often fails to distinguish between local bottlenecks and global bottlenecks. In a μI sequence of moderate-to-large length, for example, it may be the case that the insertion of a particular conflict will cause the number of μIs to be increased if added to near the beginning—but not the end—of a μI sequence. This coupling method assigns a single cost to the μOp over the entire segment, potentially causing poor code to be generated in the presence of local bottlenecks.

We conclude that *iteration coupling* is not sensitive to subtle features of the microarchitecture, features that often determine how well code compacts. We also remark that this method assumes that μI conflicts are modeled by conflict classes; this assumption is false for some microarchitectures. The consequence is not that the method will fail to work, but that it will be necessary to make some simplifying assumptions about the architecture, causing its feedback to be even less accurate.

The one positive thing we have to say about *iteration coupling* is that it does produce a number of different sequences, even if some of them them were worse than the one originally generated. As evidence that this method has some merit, we point out that it was able to discover the sequences using the constant register without requiring precompilation of the expression

```
(<- areg 0)
```

8.4. The Squeeze Method

The third and final coupling method that we tested is the *squeeze* method, given its name because the code generator is required to "squeeze" all of the μOps into a certain number of partially-filled μIs as it produces them. Originally we had planned to perform a complete

compaction each time a μOp was considered, but the cost of setting up the compaction, mapping the μOps into bundles, and compacting the code was too great to perform in an inner loop of the algorithm. Ideally, it would be nice to have an incremental compaction algorithm.

8.4.1. Modifications to code generation routine

Instead of performing the compaction each time, we approximate a compaction by keeping a count of the number of times each conflict is used. When code is to be generated, constraints such as "the *ALU* may only be used during three μIs" are specified. This is quite easy to implement: an array of integers keeps track of the number of times each conflict is used. Whenever a μOp is added, the array elements corresponding to each of its conflicts is incremented; similarly, whenever a μOp is removed—as a result of an unsuccessful search, for example—the same array elements are decremented. This "squeeze array" is used as an additional search cutoff: whenever the addition of a μOp causes the count for any conflict to exceed its limit, the μOp is immediately removed from consideration as a candidate.

8.4.2. Examples

The first example—that of putting "-2" on the *fbus*—illustrates the only success we had with the *squeeze* method. Previously, it was noted that the best method of putting a "-2" on the *fbus* was to use put a "-1" in *breg*, "0" in *areg* and to set the ALU/carry so that *breg* - *areg* - 1 is computed. Because there are a number of other solutions that do not require as many axioms to be applied, the *And/Or* and *iteration* coupling methods never found this solution. In performing this experiment with the *squeeze* method, we added the requirement that no conflict could appear in the solution more than once;[8] thus solutions found by previous methods would necessarily be pruned in this case, because each requires the use of some resource for more than one cycle.

During the first subiteration, the cutoff was small enough so that only the *AND*, *OR*, and *XOR* ALU operations—not subtraction—were considered; this search ended in failure after examining 43 nodes in the search tree. After the cutoff was increased by 30%, the search considered 5 ALU operations—including subtraction—resulting in a search that found the solution after examining 253 nodes—a number that we believe borders on being excessive.

In the second example code is to be generated for the expression

```
(; (<- dram[dadr 0] 1incwd) (<- fbus 7))
```

---

[8] We chose this restriction for the problem because we knew *a priori* that there exists a solution that satisfies it. In a full compiler, the issue of determining such "shapes" would be an issue, but we do not address it here.

in which no conflict is allowed to appear more than twice. In this case, the successful search is able to generate code without ever having to prune the search using the "squeeze" heuristic, because even the search without coupling found a sequence that did not use any conflict more than twice. The fact that optimal code is generated is therefore not indicative of the usefulness of the *squeeze* strategy.

The *squeeze* method never found a solution for the third example,

```
(; (<- gpr[2] dram[dadr 0]) (<- fbus 7))
```

In this case, code was generated first for the expression

```
(<- fbus 7)
```

which consists of the μOps that use the *fbus* twice. After that subsearch returned successfully, the search was required to find a solution to

```
(<- gpr[2] dram[dadr 0])
```

without using the *fbus* —a task that is impossible. If the order of the subsearches had been reversed, a solution could have been found rather quickly that used the constant register to generate the "7"; unfortunately, the evaluation function had no way of determining which subgoal was more "flexible".

8.4.3. Evaluation

We conclude from our experiments that this method can be of use in special situations, but that it is generally not very effective. The most fundamental problem is that the evaluation function has no knowledge about the "squeeze cutoff", and therefore guides the search in many "promising" directions that become "surprising" dead-ends. Judging from our experience, it is very important that the evaluation function be a reasonably accurate reflection of the search itself. Although this method found the optimal solution in the first example, its weakness became evident when the window was expanded to two or three μIs.

Another drawback of the *squeeze* method is that it requires the "shape" of final code to be guessed before the code is generated. For the last two examples, we also tried invoking the search routine with code space requirements that were too stringent, hoping that such searches would terminate very quickly. Unfortunately, axioms were applied profusely, and the search was time-consuming and ineffective.

Still another problem—exemplified by the third example—is that the order in which two or more conjunctive subgoals are examined can determine whether the search fails or succeeds. If a solution to the "flexible" subgoal is generated first, it is possible that no solution will ever be found because the code generator will insist on generating code for the "inflexible" subgoal that fits into an incompatible "shape". It is not clear to us that it can always be determined which of two subgoals might be more adaptable to a solution by an

alternate μOp sequence. We have not explored the possibility of rating subgoals with respect to the number of different possible code sequences they might generate.

8.5. Combining Methods

In this section we briefly describe experiments in which the *And/Or* and *iteration* methods were combined and applied to the three examples that have been used in this chapter. We found that the squeeze method was difficult to combine with either of the other two: we did not combine it with *And/Or* because conflict counting cannot be performed in a straightforward manner when multiple solutions are generated. The *iteration* method requires feedback from *successful* searches; we therefore did not combine *iteration* and *squeeze* because the "philosophy" behind the squeeze method is that the search should be so constrained that *any* solution found will fit into the minimum space. It therefore does not make much sense combine these two methods unless one of them is altered.

The *And/Or* and *iteration* methods, on the other hand, are quite easy to combine. All that is needed is to use the *And/Or* method as we normally would, and then perform the post-compaction analysis, table update, and iteration that is always done for the *iteration* method.

Although the optimal sequence for putting "-2" on the *fbus* was not discovered, the solution involving the constant was found without precompiling the

```
(<- areg 0)
```

sequence that was necessary when the *And/Or* method was used alone; in addition, a sequence using the *XOR* ALU operation was found—one that had not been found when either method was used alone. A summary of this search is given in Table 8-4.

iteration	*subiterations*	*total # nodes*	*minimum # μIs*	*comments*
(1)	1	38	2	2 solutions, using mask
(2)	2	46	2	1 solution, using mask
(3)	2	55	3	2 solutions, using constant register
(4)	3	76	3	same as (3)
(5)	2	235	??	no solution after 235 nodes

Table 8-4: Summary of first combination experiment.

The second and third examples, whose summaries are given in Tables 8-5 and 8-6, gave similar results. The use of *iteration* in addition to *And/Or* generated all of the code sequences found previously and new ones as well—all *without* the need for precompiling the "0 to *areg*" sequence.

iteration	*subiterations*	*total # nodes*	*minimum # µIs*	*comments*
(1)	1	51	3	2 solutions, using mask
(2)	2	81	5	2 solutions, using constant register

Table 8-5: Summary of second combination experiment.

iteration	*subiterations*	*total # nodes*	*minimum # µIs*	*comments*
(1)	1	109	4	10 solutions, using mask
(2)	2	94	4	2 solutions, using mask
(3)	2	141	3	4 solutions, using constant register

Table 8-6: Summary of third combination experiment.

8.6. Summary

Based on the experiments that we have performed, we must conclude that the *And/Or* method is the most effective of the three for generating code that compacts well, but that the combination of the *And/Or* and *iteration* methods appears to be even more effective. We believe that *And/Or* is the best of the three because neither of the other methods actually attempts to compact different combinations of µOps. Our experiments have convinced us that subtle characteristics of microarchitectures—timing, for example—are often critical in determining whether two sets of µOps will compact together well. Methods that do not actually attempt such compactions are likely to overlook many of these subtleties.

One problem that we have not yet resolved with the *And/Or* method is that of preventing the search from continuing to examine hundreds of nodes in the search tree looking for non-existent or highly inefficient solutions, while at the same time, giving all nodes of the search tree a "fair shake" in attempting to find alternate solutions that may lead to a better compaction. Although Cattell used a *breadth limit* to limit the search, his limit was based on the search depth. Because our search is pruned in a more flexible manner, we see no obviously "right" way of incorporating a breadth limit; still it seems that such will be necessary in order to control runaway searches.

We were disappointed that the squeeze method did not generally seem to do well, particularly since it was the only method to find the optimal solution to the first example. In retrospect, the *squeeze* method appears to apply too much "brute force", and will be applicable only in extremely "tight" situations.

Chapter 9
Conclusions

As a result of this research effort, we conclude that the code generation and compaction phases of a compiler can be coupled in such a way that microcode is produced that is of higher quality than that produced by a compiler in which the phases are executed sequentially. In addition, we believe that micromachine features make it necessary to attempt compaction on several feasible μOp sequences in order to determine which compacts into the smallest number of μIs.

In the first section of this chapter, we discuss what we believe are the major contributions of this dissertation in the area of optimizing compilers for horizontal target architectures. Following that, we discuss the limitations of our work and suggest promising areas for future research.

9.1. Contributions

We believe that the major contributions of this dissertation are:

- The development of a micromachine model that expresses both semantics and timing information in a flexible—but useful—manner.
- An extension of the code-generator generator work of Cattell [Cattell 78] with more powerful heuristics that enable successful searches at a depth approximately three times greater than the original implementation.
- A demonstration that *constant unfolding* is a useful optimization technique for horizontal target architectures.
- The discovery of a polynomial-time algorithm for optimally solving the *classical microcode compaction problem* for any real micromachine—a problem previously thought to be NP-hard—and subsequent analysis that suggests that the problem of originally ordering the μOps—previously considered secondary—is both more difficult and more important.
- The testing of three methods of coupling code generation and compaction, and the conclusion that presence of micromachine features makes it highly desirable to compact a number of different semantically equivalent code sequences before selecting the final code.

We believe that the manner in which timing constraints are specified here is significantly better than in other models we have seen because each resource is treated separately with respect to timing. Other models treat all data inputs to a given μOp identically, and therefore cannot express requirements such as an address having to be stable for one subcycle before data during a write operation.

The ability to perform successful searches in which axioms are applied at depths of ten or greater is a significant improvement over the implementation by Cattell, an implementation that itself was quite impressive. We believe that such an improvement was necessary in order to extend his algorithms to the domain of horizontal microcode; still, we often wished during our experiments that the evaluation function was yet more accurate.

The demonstration that constant unfolding is effective is perhaps the result with which we are the most pleased. Our microprogramming experience had previously convinced us that the generation of constants in the "standard manner" often results in poor-quality code. We are therefore happy to report that constant unfolding has been successfully performed, and has led to code improvement in a number of cases. The discovery that constant unfolding could be extended by applying it to subexpressions, thereby subsuming a number of *ad hoc* optimizations, is evidence that such an optimization may even be useful in compilers (or compiler-compilers) for macroarchitectures.

Perhaps the most significant result is that the *classical microcode compaction problem* does not model data relationships between μOps in a general manner, and therefore fails to acknowledge many semantics-preserving orderings of μOps. We hope that our arguments that determining the initial ordering of the μOps is the more important problem will cause researchers in the area to direct their attention towards this more challenging problem.

Finally, the original goal of our research—that of testing phase-coupling methods—has been moderately successful. We believe that we have given convincing arguments that the coupling problem should be addressed in an optimizing microcode compiler, and have presented results indicating that the *And/Or* method shows particular promise for future compilers.

9.2. Future Work

Although we believe that our research effort was generally successful, there were a number of areas that we did not have time to explore, or in which we simply failed to make headway.

Perhaps the most critical is in the area of automatically producing code that intelligently performs rotations, shifts, and bit extractions. Our evaluation function does not "understand" the semantics of such operations, and consequently the heuristic search rarely finds code

sequences that depend on such operators. One of the major problems we encountered in attempting to incorporate such knowledge into our evaluation function is that it appears that logically we need a separate *distance table* for every combination of rotation, shift, and bit length; the size of such a set of tables would be prohibitive. Cattell noted that the understanding of such operators was beyond the scope of his system; based on our reasonably intense (and extremely frustrating) effort to incorporate such understanding into our system, we consider this problem to be exceedingly difficult. Our problem is compounded by the fact that microprograms tend to perform a great deal of shifting and masking; a machine-independent microcode generation system *must* handle rotations, shifts, and bit extractions.

Another area that warrants further study is that of incorporating some sort of breadth limit into our algorithm in order to guarantee that all subsearches terminate in a reasonable amount of time. We are reluctant to adopt a strategy that makes the breadth limit a function of search depth because the current depth of a subsearch has little correlation with the amount of effort we are willing to expend in finding a solution; rather, the *search cutoff* serves that function. Our simple-minded attempts to make search breadth a function of the search cutoff have thus far not been effective.

We explored only three methods of coupling the code generation and compaction phases of the compiler. Although we had moderate success, we must certainly not rule out the possibility that some other method of coupling the phases might prove to be the most effective. In particular, methods that actually perform compaction on several code sequences seem worthy of investigation.

More generally, further work is needed in developing coupling methods among other phases of the compiler. The research of DeWitt [DeWitt 76] suggests that register allocation and compaction should be coupled. We have also argued in Section 2.2.6 that evaluation order determination is integrally tied to compaction. Additionally, several other optimization problems mentioned in Chapter 2 warrant further study.

Although constant unfolding has been quite successful, it is likely that it will not be practical to apply constant unfolding axioms at *compile time* for a production compiler. We suggest that it might be appropriate to develop techniques for analyzing a microarchitecture at compiler-compile time in order to discover "unusual" ways of producing various combinations of constants (or constant classes), storage resources, and operators, so that most of the constant unfolding work is performed only once for a given microarchitecture.

Similarly, the time required for the heuristic search to generate code may make the entire code generator impractical for production compiler. We anticipate that it will be necessary to precompile most common sequences, letting the compiler spend most of its time searching

for unusual sequences that might compact well in a particular program. Such a strategy, however, gives rise to new problems. It must somehow be decided what a "common" sequence is; research suggests that this problem is quite difficult [Cattell 78]. Furthermore, if any searching at all is done at compile time, methods must be developed for determining source expressions that warrant further searching and how much search time the compiler should spend for a particular subproblem.

As we have stated before, we—unlike many others—do not believe that the *intrablock* compaction problem is solved. Further research is necessary to develop compaction algorithms that consider partial orders other than the one implied by the ordering of the μOps that are passed to the compaction phase. This will certainly be true in a production compiler, where the μOps are passed to the compaction phase in the form of a graph rather than as a sequential list.

We suggest that dynamic programming may prove to be useful in compacting microcode, particularly after the ordering of register usage has been determined. Although the complexity of the *chain matrix compaction algorithm* is, in theory, a polynomial whose degree is the number of registers in the micromachine, we suspect that in practice the complexity will be much lower if the algorithm is optimized so that it does not create portions of the matrix-graph that are subsequently removed. In addition, preliminary study indicates that dynamic programming shows promise for compacting tight loops.

Finally, the *classical microcode compaction problem* contains several other scheduling problems as special cases. It may therefore be worthwhile to apply it to other situations in which the "breadth" of the partial order is small.

Appendix A
Deterministic Code Generation Algorithm

This appendix discusses in detail the ordering and pruning mechanisms that allow the code generation algorithm to run on a deterministic machine. Because the evaluation function is so complex, we treat it separately in Appendix B; for the purpose of this discussion, the reader can assume that the evaluation function compares two operands and returns a value that represents the cost of transforming the first into the second.

Research in artificial intelligence has demonstrated that a depth-first searching strategy is highly dependent on the order in which the nodes of the search tree are examined, while a breadth-first searching strategy is not [Nilsson 80]. If a depth-first strategy is used, it is possible for an enormous amount of time to be spent searching down dead-end paths of the search tree, even when a shallow solution exists. A breadth-first search is guaranteed to find a shallow solution before it finds a deep one.

Although a breadth-first search appears to be attractive, it is probably not practical:

- In a breadth-first search, all nodes are expanded in parallel; thus the search requires an amount of space that is exponential with respect to its depth. A depth-first search requires only linear space.
- The search depth should not be defined by the number of nodes examined, but rather by the cost of the μOps generated along the path. If this is the case, then the application of an axiom during the search would not increase the "depth" of the search. This could give rise to arbitrarily long paths of depth zero in the search tree. For example, the repeated application identity axiom could lead to the path:

  ```
  x => (+ 0 x) => (+ 0 (+ 0 x)) => ...
  ```

  Clearly, a search that expands such a path until its cost became non-zero would be ineffective.
- The *most shallow* solution is not necessarily the *least expensive*; the *cost* of a *AND* node in the search tree is the sum of the costs of its sons rather than their minimum. The two *And/Or trees* in Figure A-1 demonstrate this; the depth of a solution to the tree on the left is 5, but the total cost is 45 because the *AND* node requires that the costs be summed. Conversely, the depth of a solution to the tree on the right is equal to its cost, 10.

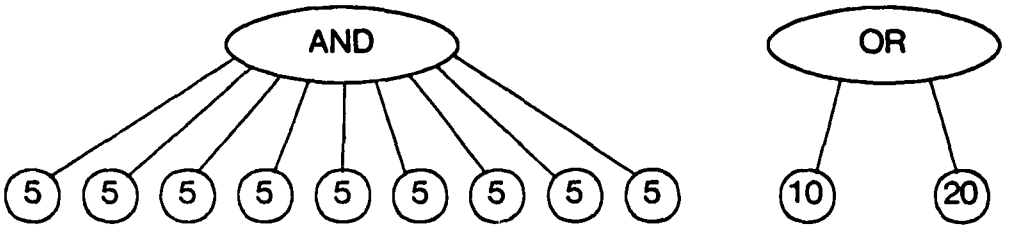

Figure A-1: Two *And/Or trees* with different costs.

We use the *iterative deepening* [Slate 77] technique to approximate a breadth first search. First, a depth-first search is attempted with with a shallow depth limit. If no solution is found, the search is repeated with progressively greater depth cutoffs until a solution is found. In addition, we have added a caching mechanism, which has proven useful in pruning the search in a several ways.

The remainder of this appendix is organized as follows. First, the data structures used by the deterministic algorithm are described. Next follows by a detailed discussion of the basic searching strategy. Then descriptions are given of additional mechanisms for limiting the search breadth. Finally, an example is presented, illustrating how the pruning and ordering mechanisms work.

A.1. Data Structures

The deterministic search algorithm uses two data structures in addition to those used by the nondeterministic algorithm. The first is a table that defines a cost for each μOp. The second is a cache that stores the results of previous searches. The μOp cost table is a one-dimensional array that specifies an integer cost for each μOp; as was discussed in Chapter 5, the cost of a μOp is initially computed by summing the cost of the conflict classes to which it belongs.

As μOps are generated during the heuristic search, the sum of their costs (which defines the search depth at any given node in the search tree) is accumulated. If the depth along a search path exceeds a preset limit, the search path is pruned.

The cache, which records the results of all previous calls to *search* and *transform*, contains two fields for each entry:

- A *cache cutoff*, which is the greatest depth at which a *search/transform* has been attempted with a particular set of arguments.
- A result, which is a tree of μOps that resulted from the search at that depth.

The cache is used to prune the search in several ways, and is discussed further in Section A.2.5.

A.2. The Algorithm

The code generation algorithm is a further specification of the nondeterministic algorithm presented in Chapter 6, and resolves the following questions:

- At what cost (depth) should the search be attempted at the top level? What action should be taken if no solution is found?
- How should the search be bounded? In other words, how should it be decided that a path is no longer worth pursuing?
- In what order should the nodes be examined?
- How should the cost be allocated when a search is decomposed into several subsearches?

In this discussion, we assume that the code generation algorithm is satisfied with a single solution, and therefore terminates the search when it finds a solution. Extensions that allow the search to generate multiple solutions are discussed in Chapter 8.

A.2.1. Search cutoff

The primary method of pruning the search is through the use of a *search cutoff*; whenever *search* (or *transform*) is called, it is passed a *cutoff* that specifies the cost above which a solution is unacceptable. Any search path is immediately pruned that would, according to the evaluation function, exceed the cutoff; thus only paths that "show promise" are pursued.

The cutoff is normally passed without change down the search tree. In two instances, however, the cutoff is modified. First, the cutoff is divided among subsearches when a search is decomposed (see A.2.3). Secondly, whenever a μOp is selected on a particular search path, the cost of the μOp is subtracted from the cutoff.

A.2.2. Beginning the search

When the code generator is invoked to produce code for a particular expression, the evaluation function estimates the cost of producing of code for that expression. The initial cutoff is determined by multiplying this estimate by a prespecified constant (e.g., 1.25) in order to account for the fact that the evaluation function is often too optimistic in its estimates.

If the search with the initial cutoff is unsuccessful, it is increased—again by multiplying by a prespecified constant—and the search is retried. This process is continued iteratively until either a solution is found or a time limit is exceeded.

A.2.3. Allocating costs among sub-searches

There are a number of circumstances in which a search is decomposed into subsearches.

If *expr1* is divided into *expr2* and *expr3* during a search whose cutoff is 100, we must determine the values *x* and *y* in

```
search(120): expr1
  decompose search:
    search(x): expr2
    search(y): expr3
```

In this case, it is necessary to determine new cutoffs for each of these subsearches. During the course of our research, we have tried four different methods for determining such cutoffs:

1. Pass the cutoff directly to each subsearch. The values for *x* and y would then both be 120 in the above example.
2. Use the evaluation function to determine minimum requirements for the search, and divide the "slack" evenly among the subsearches. Assuming that the evaluation function "rated" *expr2* at 40 and *expr3* at 30, *x* and *y* would then be 65 and 55, respectively, the slack of 50 being divided evenly between *expr2* and *expr3*.
3. Divide the cutoff so that each subsearch receives slack in proportion to its evaluation function rating. In this case, the cutoff for *expr2* and *expr3* would be 68.6 and 51.4, respectively.
4. Divide the cutoff so that each subsearch receives slack in proportion to the *square* of its evaluation function rating. In this case, the cutoff for *expr2* and *expr3* would be 72 and 48, respectively.

The last three of these methods have the advantage that they guarantee that the total cost of μOps will be less than the cutoff, and will prune the search more quickly if the evaluation function has been overoptimistic. Although method 3 might in some sense seem the "fairest", we have found that 4 is the most effective. It appears that this is because the evaluation function is most accurate when its result is small, so a policy of assigning most of the slack to those expressions whose evaluation function is large accounts in some manner for the fact that those expressions probably need more slack due to an inaccurate estimate by the evaluation function. An exception to this policy occurs in the case where a search is decomposed and the sequencing operator (;) is the outermost operator. In this case, the searches are really independent, and the slack is distributed proportionally (i.e., method 3).

A.2.4. Node ordering and selection

There are a number of points in the search where a *nondeterministic* choice must be made. In the *transform* function for example, it is possible that several axioms and constant unfolding axioms and the operand-by-operand decomposition are all applicable. In such a case, the evaluation function is used to rank each potential choice. The lowest-valued choice is attempted first, then the second, third, and so forth, until either the search completes successfully, or all choices have been exhausted. In the former case, *search/transform* returns successfully; in the latter, unsuccessfully.

A.2.5. Caching search results

We have found that the evaluation function alone does not adequately bound the search, and have therefore added a caching mechanism. The result of each call to *search* or *transform* for a given set of arguments is recorded, along with the highest cutoff value with which it was called. The cache is used for pruning the search in three situations:

- When a search is attempted on a result for which a prior result exists that satisfies the cutoff criterion, the previously computed result is used immediately.
- When an identical (unsuccessful) search has already been completed with a cutoff whose value is greater than or equal to the present cutoff, the local search is immediately terminated.
- When an identical search is already in progress, the search is terminated immediately. This often happens when a search calls itself indirectly as a result of the application of two or more axioms that "cancel each other out" (e.g., two commutative axioms applied consecutively).

The *transform cache* is also used by the evaluation function; this will be discussed in Appendix B.

A.3. Limiting Search Breadth

In addition to using the evaluation function and cache for pruning the search, we have introduced a number of other rules for limiting the breadth of the search. The first rule requires that a feasible μOp whose semantics are defined by an assignment statement have the same destination operand as the goal (not counting indices). This avoids a great deal of redundancy resulting from the the selection of μOps in different orders during the search. For example, the solution

```
(<- b a)
(<- c b)
(<- d c)
```

of `(<- d a)` could be discovered in five different orders by the heuristic search. With the "matching destination" rule, only one of these orderings is considered.

The other three rules for limiting search breadth are included as a result of experiments that led us to conclude that the application of axioms often causes the search breadth to increase in an unmanageable manner. First, an axiom may be applied only if it causes the outermost operators of the new expressions to match. Secondly, an axiom or constant unfolding axiom may not be applied if introduces an operator that is not already present in either the *goal* or *current* expression. Finally, the total number of axioms and constant unfolding axioms applied at any node in the search may not exceed a predefined limit, which is a function of search depth (in terms of number of axioms applied), and was introduced after experiments

revealed that the eager application of axioms often causes enormous amounts of time to be spent following "ridiculous" paths.

Pruning mechanisms carry with them the danger that branches leading to good solutions might also be lost. This has in fact happened during our experiments, but we see no way of avoiding it. Unless the evaluation function is perfect or an exhaustive search of the solution space is feasible, we must accept the fact that some good solutions will be missed.

A.4. Specification of the Algorithm

We are now ready to present the deterministic version of the code generation algorithm.

Search(goal) =

1. If a failure is found in the *search cache*, and the *cache cutoff* is as least as large as the *search cutoff*, return a *failure*.
2. If a success if found in the *search cache*, and the *search cutoff* as least as large as the *cache cutoff*, return the result from the cache.
3. Otherwise, mark the cache entry as a failure (so that this call to *search* will not directly or indirectly call itself with an identical argument) and use the evaluation function to select the decompositions (for sequencing, iteration and looping operators) and feasible μOps that have values less than the *search cutoff*. (Feasible μOps whose definitions are assignment statments must have destinations that match the destination of the goal. Furthermore the cost of such a μOp is added to the value of evaluation function.) Then in order of evaluation function rating, perform the following to each decomposition or feasible μOp until a successful search is found or all selected feasibles and decompositions have been tried:
   - If the selection is a feasible μOp, *transform* on the respective sources and destinations, with the cost of the μOp being subtracted from the cutoff. If the outermost operator is an assignment, the transformation between the destination operators is reversed, and the *reverse index flag* is set.
   - If the selection is a decomposition, the search is decomposed into its component parts. If the outermost operator of the *goal* is a sequencing operator, data dependency links are added between certain references to resources in the original expression. If the *goal* expression is a conditional or iteration, new flow graph nodes and links are generated.

   In all cases, the cutoff is divided among the *search* and *transform* functions in the manner described in Section A.2.3.
4. Finally, the *search cache* is updated to reflect the result of this call to *search*.

Transform(goal, current) =

1. If a failure is found in the *transform cache*, and the *cache cutoff* is as least as large as the *search cutoff*, return a *failure*.
2. If a success is found in the *transform cache*, and the cost *search cutoff* as least as large as the *cache cutoff*, return the result from the cache.
3. If the operands are identical or if *goal* is the *undefined resource*, return an empty list, signifying that no μOps are necessary to transform the first operand into the other. If the operands are identical constants or resources, place a data dependency link between *goal* and *current*; if the operands are identical expressions, recursively call *transform* on each pair of suboperands.
4. If *current* is a constant pattern, and *goal* is a "compatible" literal constant or constant pattern, place a data dependency link between *goal* and *current*, and create and return a pseudo-μOp whose operand is *goal*.
5. If both expressions are identical storage resources, but with non-identical indices, apply *transform* to the indices; if the *reverse index flag* is set, reverse the sense of the transformation.
6. If *current* is a storage resource, and step 5 does not apply or did not succeed, apply the fetch decomposition:

   ```
   search: (<- current goal)
   ```

   Otherwise, mark the cache entry as a failure (so that this call to *transform* will not directly or indirectly call itself with identical arguments) and use the evaluation function to select axioms and constant unfolding axioms that result in *goal* expressions that are "rated" below the cutoff value, eliminating any that fail to satisfy the criteria of Section A.3. If the outermost operators of *goal* and *current* are identical, and the operand-by-operand decomposition is rated below the cutoff, also include it in the list of feasible axioms. Then in order of evaluation function rating, with the decomposition taking precedence if there is a tie, perform the following to each decomposition or axiom until a successful search is found or all selected axioms and decompositions have been attempted:

   - If an operand-by-operand decomposition is selected, call *transform* recursively on an operand-by-operand basis, returning all μOps generated by any of the calls.
   - If an axiom or constant unfolding axiom is selected, apply it to the goal and attempt to *transform* the modified *goal* into *current*.

   In all cases, the cutoff is divided among the search and transform functions in the manner described in Section A.2.3.
7. Finally, the *transform cache* is updated to reflect the result of this call to *transform*.

A.5. An Example

As an example of the algorithm in action, let us consider a problem on the Puma micromachine [Grishman 78]. (A description and sketch of the Puma may be found in Appendix E). The problem is to add the constant 5 to the *buffer* register, and to store the result in the *AC* register. The problem is especially interesting because the Puma has two ALUs: an exponent ALU (EALU) and a normal ALU. The literal field of the μI is directly connected only to the former, while the *buffer* register is directly connected to the latter; the presence of two ALUs, neither of which is "obviously" the right one to use, makes the job of discovering the best code sequence more difficult.

The initial call, with a cutoff of 69.60,

```
search(69.60): (<- ac (+ 0000005 buffer))
```

is followed by a few calls to *search* and *transform* that discover μOps (with a total cost of 5) that move the final answer from the *ALUX* register to the *AC*. At this point, the problem has been reduced to

```
search(64.60): (<- alux (+ 0000005 buffer))
```

and a decision must be made about which ALU should be used. From the perspective of the heuristic search, the decision takes the form of deciding which of the feasible instructions

`alux.or = (<- alux (or alu e0))` *or* `alux.alu = (<- alux alu)`

should be selected next. The evaluation function predicts that *alux.or* is likely to be less expensive, so it is selected and the "*OR* identity" axiom is applied, resulting in the call

```
transform(64.60): (or 0000000 (+ 0000005 buffer)) => (or alu e0)
```

This, in turn, results the operand-by-operand decomposition,

```
transform(2.02): 0000000 => alu
```

and

```
transform(62.58): (+ 0000005 buffer) => e0
```

with most of the cutoff value being assigned to the latter task. A μOp that computes a zero in the ALU is found immediately, but after expending a moderate amount of effort, the search for a solution to the latter task returns with failure; as it turns out, it is impossible to move the value of the *buffer* register unmodified to an EALU input.

After this failure, the search backtracks to the point where the *alux.alu* μOp is considered. This leads to the selection of a μOp

```
alu.plus = (<- alu (+ (+ ac buffer) carryin))
```

which results in the call

```
transform(62.60): (+ 0000005 buffer) => (+ (+ ac buffer) carryin)
```

After applying the additive identity axiom and finding a μOp that sets *carryin* to zero, the problem is reduce to that of transforming the constant "5" into *AC*. Again the μOps that move the value of *ALUX* to *AC* are easily discovered, so the problem becomes

```
search(57.60): (<- alux 0000005)
```

Again *alux.or* is selected ahead of *alux.alu*. This time, however, the subproblems become (after the "OR identity" axiom is applied)

```
transform(2.20): 0000000 => alu
```
and
```
transform(55.40): 0000005 => e0
```

The solution to the first of these problems is read from the cache; the second results in the μOp

```
ealu.plus = (<- ealu (+ ea eb))
```

being selected (after finding the μOp that moves data from *ealu* to *e0*). This, reduces the problem to transforming the constant 5 into the sum of the two ALU inputs:

```
transform(52.40): 0000005 => (+ ea eb)
```

In this situation, the author expected the additive identity axiom to be applied, and a zero to moved to one input from another part of the machine. Instead, a constant unfolding axiom was applied that allowed the "−1"—which is directly connected to one of the EALU inputs—to be used; thus the code that was discovered set the literal field to "6", and added it to the "−1" causing a "5" to be produced, thereby completing the search.

The entire search examined 63 non-trivial nodes in 48.33 seconds, with a maximum search depth of 28 nodes, and a maximum depth in applied axioms of 4. The resulting code is:

```
ea.con 6     load constant 6 into "A" input of EALU
eb.ones      load all ones into "B" input of EALU
ealu.plus    perform an addition in EALU
ld.e0        load register E0 with the output of EALU
alu.0        set ALU function to "zero"
alux.or      "OR" constant 5 with the zero from ALU output
shlo.pass    pass constant 5 through shifter without shifting
ac.lo        load the AC with the constant 5 from shifter
carry.0      set the ALU carry input to 0
alu.plus     add the values in the BUFFER and AC together
alux.alu     do not "OR" the value of E0 with ALU output
shlo.pass    pass final result through shifter without shifting
ac.lo        load the AC with the final result from shifter
```

Appendix B
The Evaluation Function

This appendix describes the evaluation function that is used to guide the heuristic search. It is our hope that someone who understands its contents will be able to reproduce (and probably improve upon) the code generator; there are therefore necessarily many details. A casual reader may wish to ignore this appendix altogether.

Nilsson [Nilsson 80] claims that the evaluation function is a critical component of any heuristic search. We certainly agree with his assessment; More time was spent testing and modifying the evaluation function than any other single component of the microcode generation system because the entire search depends on its estimates being reasonably accurate.

The evaluation function in our system compares two expressions and estimates the cost of transforming the first into the second. It is important that the evaluation function take into account the overall searching strategy, the μOps available on the target architecture, and the axioms that are available for performing transformations. The success of code generation process is largely dependent on the accuracy with which the evaluation function reflects the heuristic search.

The evaluation function makes use of a number of *distance tables*, which contain estimates of the cost of transformations or data movements between storage resources, operators and constants. When two atomic operands (a storage resource or constant) are compared, the evaluation function generally performs a table lookup. When one or both of the operands is an expression, portions of the expression are compared in different combinations to arrive at an estimate of the "distance" from one expression to another. This generally involves recursive calls to the evaluation function; the *distance tables* are therefore ultimately used in all cases.

In order to increase the efficiency of the evaluation function, we have introduced a *cutoff* parameter, which allows the computation to be terminated early in many cases. The cutoff is useful because it is often the case that the *search* and *transform* functions are only interested in a solution whose value is below a certain threshold. In such cases, the evaluation function

computation is terminated as soon as it determines that its value is above the cutoff threshhold. Measurements suggest that the use of this cutoff increases the speed of the evaluation function by about a factor of two.

The remainder of this section is organized as follows: Some preliminary definitions are given, followed by a description of the data structures that are used. Then, the algorithm itself is described, followed by detailed examples. Finally, the evaluation function is analyzed in terms of its effectiveness, with particular emphasis on its known shortcomings.

B.1. Some Definitions

Before discussing the evaluation function itself, we wish to define a few terms that will be used throughout the section. For these definitions, we will assume that *X* and *Y* are expressions as defined in Section 5.2.2, and that *E* is the expression

```
(+ 4 (and %mask (rotate abus regfile[23])))
```

The first few definitions are quite simple. *Atoms*(*E*) represents the set of all atomic operands of *E* (i.e., storage resources and constants, excluding indices): "`4`", "`%mask`", "`abus`" and "`regfile`". *SubOpds*(*E*) are the top-level operands of *E*: "`(and %mask ...)`" and "`4`". *Operators*(*E*) are the operators in *E*: "`+`", "`and`" and "`rotate`", while *Size*(*E*) is the total number of operators and atoms in *E*, excluding indices, which in this case is *seven*. Finally, the *outermost operator* of an expression is the operator in the leftmost position as it is itten; *OuterOp*(*E*) is "`+`".

The other terms deal with properties of the operators themselves, or define data structures used by the evaluation function. The *index cost* of an indexed storage resource (e.g., `regfile[23]`) is the cost of transforming the actual index (e.g., 23) into an operand that actually indexes the resource in a μOp definition. Thus, if there were a μOp with the semantics

```
(<- abus (regfile [regidx]))
```

then *IndexCost*(*regfile*[23]) would be the cost, as estimated by the evaluation function, of transforming 23 into *regidx*. If more than one such expression occurs in the μOp definitions, the smallest value is used.

The *table cost* between two operands/atoms, denoted $X \rightsquigarrow Y$, is the cost of transforming or moving the first to the second as determined by a table lookup. A discussion of the tables may be found in Section B.2.1.

The *data operands* of an expression are those suboperands for which the operator may act as an identity operator, given the proper values for the other suboperands. This information is used by the evaluation function in estimating how data may be routed. For example, both

operands of the "+" operator are data operands, as zero may act as either the left or right identity. The second (but not the first) operand of the "rotate" operator is a data operand because *rotate* has a left identity but no right identity.

The *identity cost* of an operator is the difficulty, according the evaluation function, of transforming the operator into the *identity operator*, and is found by table lookup, *ident↝op*. The *identity depth* of one expression within another is the sum of the *identity costs* of all operators that are ancestors of the first expression in the second. It is an estimate of the cost of transforming the first expression into the second by the application of identity axioms.

The *μOp expressions* of an operator are those expressions occurring in the μOp definitions that contain either the operator itself, or a "closely related" operator. the evaluation function uses these expressions to determine whether a particular operation can be performed anywhere on the micromachine.

Finally. we define the *axiom factor*, a "fudge factor" that is used to account for the fact that an axiom often brings new operators and operands into the search. In transforming `(not A)` into B, for example, one may have to account for the fact that the axiom

```
(not $1) :: (xor -1 $2)
```

introduces a new operator, *XOR*, and new literal, "-1". The axiom factor is a very rough estimate of the the extra μOps that are necessary to generate these new additional constants and operators. The axiom factor is defined as a percentage (currently 14%) of the cost of the entire μl (i.e., the sum of the costs of all conflicts) and is used to by the evaluation function to multiply costs involving operator comparisons.

B.2. Data Structures

The evaluation function uses several data structures in performing its task. As was mentioned earlier, there are a number of tables which estimate the distance between constants, resources and operators. In addition to these tables, the evaluation function makes use of a cache of previous results, lists of expressions involved in indexing resources, and certain information about operators, such as which ones are commutative.

B.2.1. Distance tables

Five *distance tables* are used, four of which contain estimates of the cost of transforming/moving some quantity to a storage resource. The *resource-resource table* specifies the cost of moving data from any (storage) resource to any other. The *operator-resource table* gives the cost of performing a particular operation, and then moving the data from that operation to the specified resource. The *literal-resource table* specifies the cost of moving "commonly used" literals to the resource—in our implementation, such literals

are defined to be the integers −1, 0, and 1. Finally, the *pattern-resource table* defines the distance between any constant pattern and a particular resource. The other distance table is the *operator-operator table*, which defines a how closely related a pair of operators is. Sample distance tables are given in Section B.4.1.

Once the values contained in these tables are computed, they remain fixed until the micromachine definition or axioms are changed. The *operator-operator table* is computed in four steps:

1. Initially the cost of each distance in the table is set to *infinity*, except that the distance between an operator and itself is set to zero.
2. The "cost" of each axiom is computed by counting the number of operators and constants it introduces.
3. The distance from one operator to another is the minimum axiom cost in which the first operator occurs on the left side, and the second occurs on the right side. The distance from the *identity operator* to any other operator is computed by considering axioms of the form

   $1 :: (*op opd1 opd2*)

   to be

   (ident $1) :: (*op opd1 opd2*)
4. A transitive closure is taken on the entire table.

The "distance" from one operator to another is defined to be the product of their table value and the *axiom factor*.

The *resource-resource*, *literal-resource*, *operator-resource*, and *pattern-resource* tables are determined by considering all μOps whose semantics are defined by an assignment statement. The distance to the destination resource from any other resource (or literal, pattern, operator) is computed by adding the cost of the μOp and the *identity depth* of the latter.

After the table entries have been computed, transitive closures are taken with the *resource-resource* table to account for literals, patterns, operation resources that must pass through intermediate resources. In addition, a transitive closure is taken on the *resource-resource* table with respect to the *operator-resource* table to account for the application of axioms during the heuristic search.

Conceptually, there is one more table, the *literal-pattern table*, which contains the "distance" from each literal and each pattern. This "table" is implemented in the code, however, in order to save space—a 16-bit machine with 4 patterns would require $4 \cdot 2^{16}$ table entries otherwise. For each pattern there exists a routine which determines whether a literal *matches*, *almost matches*, or *fails to match* it.

B.2.2. Caches

In order to take advantage of the fact that the same expressions tend to be repeatedly compared during a given search, the evaluation function maintains a distance cache, in which previously computed values may be looked up rather than recomputed. The *transform cache* is also used for operand pairs on which *transform* has already been called; when such a cache entry is available, the evaluation function returns an exact value instead of an estimate.

B.2.3. Other data structures

Several other data structures are used in addition to the distance tables and caches. The *index table* contains for each indexed storage resource a list of expressions that appear as indices for that resource in the μOp definitions. It is used to determine the *index cost* of an operand.

The *operator-expression table* contains the *μOp expressions* for each operator, and is used to determine a lower bound on the least expensive way to compute a given expression. The *commutativity* and *associativity vectors* are bit vectors that specify whether a given operator is commutative and/or associative, and are computed by examining the axioms. Finally, the *data operand table* specifies for each operator which of its operands are *data operands*.

B.3. The Evaluation Function Algorithm

We are now ready to present the algorithm itself, which computes a "distance" from one operand/operator to another. We use the word *distance* loosely here because it is unidirectional; it is used in the rest of this section for lack of a better term.

The evaluation function is actually a synthesis of three different functions. The *distance function* (DF) compares suboperands and operators recursively and in different combinations. The *associative distance function* (assocDF) compares operators and atomic operands, without regard for the structure of either expression. The *size-based distance function* (sizeDF) is a function of the difference in the number of nodes in the expression tree. The evaluation function is computed by taking the larger of the size-based distance function and a weighted sum of the other two:

$$EF = Max(sizeDF, Min(DF, 0.9 \times assocDF + 0.1 \times DF))$$

The purpose of the weighting between the associative distance function and the distance function is to break ties, which are often generated by the associative distance function.

B.3.1. The distance function

The distance function first checks the transform cache, returning the cost of the transform if it finds that a successful transform has been attempted. If it finds an unsuccessful transform with a high enough cutoff, it also returns returns a lower bound on the cost of the transform, which it reads from the cache. If a result cannot be inferred from the transform cache, the *distance cache* is checked. If no entry is found in the distance cache, the computation depends on the types of operands that are being compared:

If the second operand is a constant, the first operand must be a "compatible" constant:

- **Literal constant => literal constant.** If the values are equal, their distance is 0. If their values are "almost equal", which for our purposes means that the former can be converted to the latter by adding or subtracting 1, or by complementing or negating, their distance is defined to be a predefined positive integer—currently ten times the *axiom factor*—signifying that the constants are "close"; otherwise the distance is infinite.
- **Literal constant => constant pattern.** If the literal matches the constant, the distance is zero. If it "almost" matches, the distance is the predefined constant described above; otherwise their distance is infinite.
- **Constant pattern => constant pattern.** The distance is either zero or infinite, depending on whether the first pattern is a subset of the second.
- **Anything else => literal constant or constant pattern.** The distance is defined to be infinite.

If the first operand is a constant or storage resource, and the second is something other than a constant, the distance tables are used:

- **Literal constant or constant pattern => resource.** The *pattern-resource table* is examined to determine the smallest distance to the resource from any pattern that matches the first operand. If the first operand is a literal constant between –2 and 2, the literal-resource table is also used to further minimize the value. The *index cost* of the second operand is also added.
- **Resource$_1$ => resource$_2$.** The *resource-resource table* is used to estimate the cost of moving data from resource$_1$ to resource$_2$; the *index cost* of each operand is then added.
- **Literal constant, constant pattern or resource => expression.** The minimum over all atomic operands in the expression is taken of the distance from the first operand to the given atomic suboperand plus the *identity depth* of the atomic suboperand. If the expression evaluates to a constant, it is folded before the comparison.

$$\mathbf{Min}_{e \in expression} \; DF(opd_1, e) + IdentDepth(e)$$

When the first operand is an expression, the computation is dependent on its outermost operator and the *type* of second operand:

- **(Flow *opd*) => anyOperand.** When only a flow result is being passed, the value computed is the smallest distance from any *atomic suboperand* of *opd* to the second operand.

$$\underset{op \in Atoms(opd)}{\mathbf{Min}} \; DF(op, anyOperand)$$

- **Expression => resource.** When the first operand is an expression and the second is a resource, lower bounds on the distance from *expression* to *resource* are computed in two ways; the value returned as the distance is the largest of these lower bounds. The first lower bound is computed by computing the distance from each atomic operand in *expression* to *resource*, adding it to its *identity depth* in *expression*, and selecting the largest such sum.

$$\underset{a \in Atoms(expression)}{\mathbf{Max}} \; (a \rightsquigarrow resource) + IdentDepth(a)$$

The second bound is computed by finding the smallest distance between *expression* and any μOp *expression* of the outermost operator of expression, and adding it to the distance from latter to *resource*.

$$OuterOp(expression) \rightsquigarrow resource + \underset{e \in MuopExprs(OuterOp(expression))}{\mathbf{Min}} \; DF(expression, e)$$

- **(<- dst_1 src_1) => (<- dst_2 src_2).** The distance from src_1 to src_2 is added to the distance from dst_2 to dst_1.

$$DF(src_1, src_2) + DF(dst_2, dst_1)$$

- **Expression$_1$ => expression$_2$.** If the outermost operators are identical, the distances are added together on an operand-by-operand basis.

$$\sum_{j \in operand\ index} DF(SubOpds(expression_1)_j), SubOpds(expression_2)_j))$$

If the operator is commutative, an attempt is made to reduce this amount by performing the computations with the operands reversed. If, on the other hand, the outermost operators differ, the sum of the minimum distances between each suboperand of $expression_1$ and any suboperand of $expression_2$ is added to the table distance between the operators.

$$OuterOp(expression_1) \rightsquigarrow OuterOp(expression_2) + \sum_{x \in SubOpds(expression_1)} \underset{y \in SubOpds(expression_2)}{\mathbf{Min}} \; DF(x, y)$$

Whether the outermost operators are identical or not, an alternative computation is used when smaller than the above: the minimum of the distance from $expression_1$ to any suboperation of $expression_2$ plus the *identity depth* of the latter.

$$\underset{x \in SubOpds(expression_2)}{\mathbf{Min}} \; IdentDepth(x) + DF(expression_1, x)$$

B.3.2. Associative distance

The associative distance function computes an "alternate distance" between two expressions. Although we use the term *associative*, its purpose is more or less to compute distances between all operators in the expression and between all resources/constants in the expression without regard to parenthesization or order. Thus, it also accounts for other axioms, such as distributive ones.

The *associative distance* between two *assignment* statements is simply the sum of the associative distances between their corresponding operands, with the direction reversed for the destination operands. Otherwise the associative distance from one operand to another is the sum of four quantities:

1. The sum of the minimum distances from each "difficult" operator to any resource in the second operand is computed. A "difficult" operator is one that appears in the opd_1, but not in opd_2, and cannot be removed from the first by the application of an axiom without introducing additional operators.

$$\sum_{o \in difficult} \; \min_{r \in Atoms(opd_2)} o \rightsquigarrow r$$

2. The maximum of the minimum distances from each resource or constant in opd_1 to any resource or constant in opd_2.

$$\max_{x \in Atoms(opd_1)} \; \min_{y \in Atoms(opd_2)} x \rightsquigarrow y$$

3. The difference in size between opd_1 and opd_2, multiplied by the axiom factor.
4. A predefined constant, currently five times the *axiom factor*, to account for the fact that the associative distance function ignores structure, and would therefore tend to dominate other distance computations.

B.3.3. Size-based distance

The purpose of the size-based distance computation is to introduce a penalty when the size of the two operands differs greatly. It is computed by multiplying by the difference in size of the two expressions by the axiom factor.

$$AxiomFactor \times |Size(opd_1) - Size(opd_2)|$$

B.4. Examples

In this section, a simple hypothetical micromachine is described, the associated distance tables are presented, and a few examples are given to demonstrate how the evaluation function works.

B.4.1. Sample micromachine

Table B-1 shows the expression for each μOp in the hypothetical machine along with its cost,

`(<- areg gpr[%wild])`	*cost 4*
`(<- areg fbus)`	*cost 2*
`(<- breg (and %mask fblatch))`	*cost 5*
`(<- breg %wild)`	*cost 5*
`(<- fbus (+ areg breg))`	*cost 4*
`(<- fbus (- areg breg))`	*cost 4*
`(<- fbus (and areg breg))`	*cost 4*
`(<- fbus 0)`	*cost 4*
`(<- gpr[%wild] fbus)`	*cost 2*
`(<- fblatch fbus)`	*cost 1*

Table B-1: μOp expressions.

while Table B-2 shows the relevant portion of the *operator-operator table*, derived from the axioms in Appendix C. In this case, the table values are estimates of the "similarity" of two operators.

	and	*+*	*−*	*ident*
and	**0**	**∞**	**∞**	**∞**
+	**∞**	**0**	**1**	**∞**
−	**∞**	**2**	**0**	**∞**
ident	**2**	**2**	**3**	**0**

Table B-2: Operator-operator table.

The remaining tables assume that the *axiom factor* is two (2), implying that the "distance" between a pair of operators is twice the table entry. Table B-3 is the *resource-resource table*; the entries with an asterisk (*) are those derived directly from the μOps; the remaining entries were computed by the transitive closure.

Table B-4 is the *operator-resource table*, B-5 is the *literal-resource table*, and B-6 is the *pattern-resource table*.

The *index table* for the machine contains a single entry, **`%wild`**, for the *gpr* resource. The operator-expression table contains entries for three operators:

	areg	*breg*	*fbus*	*gpr*	*fblatch*
areg	0•	18	8•	10	9
breg	10	0•	8•	10	9
fbus	2•	10	0•	2•	1•
gpr	4•	22	12	0•	13
fblatch	19	9•	17	19	0•

Table B-3: Resource-resource table.

	areg	*breg*	*fbus*	*gpr*	*fblatch*
and	6	5•	4•	6	5
+	6	14	4•	6	5
–	6	14	4•	6	5
ident	8	7	6	8	7

Table B-4: Operator-resource table.

	areg	*breg*	*fbus*	*gpr*	*fblatch*
–1	15	5•	13	15	14
0	6	12	4•	6	5
1	15	5•	13	15	14

Table B-5: *Literal-resource* table.

	areg	*breg*	*fbus*	*gpr*	*fblatch*
%wild	15	5•	13	15	14
%mask	15	5•	13	15	14

Table B-6: Pattern-resource table.

and	`(and %mask fblatch) (and areg breg)`
+	`(+ areg breg) (- areg breg)`
–	`(- areg breg) (+ areg breg)`

B.4.2. Examples of the evaluation function in action

Let us consider the distance from

`(+ 3 fblatch)` *to* `(+ areg breg)`

on the machine just described. This is computed as specified in Section B.3:

1. The sum of the operand-by-operand distances is 24. The distance from "3" to *areg*, 15, is found in the *pattern-resource table*; "3" matches both `%wild` and

%mask, so the minimum distance is chosen—in this case they are identical. The distance from *fblatch* to *breg*, 9, is found in the *resource-resource table*.

2. Because "+" is commutative, the computation is also considered with the operands reversed. The distance from "3" to *breg* is 5, while the distance from *fblatch* to *areg* is 19. Again, the total is 24.

3. Next an attempt is made to use "+" in the second operand as an identity operator. Its *identity cost* (4) is added to the distance from

   (+ 3 fblatch) *to* **areg**

   which is 23, resulting in a total of 27.

4. The same is also attempted with the other operand:

   (+ 3 fblatch) *to* **breg**

   resulting in a distance of 30, and a sum of 34.

5. Finally, the associative distance is attempted; in this case, the computation is quite simple because there are no "difficult" operators, and the expression sizes are identical: 10 (i.e., five times the *axiom factor*) is added to 9, the max/min distance between atoms in the first/second operands, giving the result 19.

The distance function result is 24, the minimum of the first 4 computations. Because the associative distance is smaller, the final result is 90% of 19 plus 10% of 24, or 19.5; the size-based distance does not affect the result in this case because the expression sizes are identical.

Next, consider a similar problem, the distance from

(+ 3 fblatch) *to* **(- areg breg)**

In this case, the outermost operators are different, so different computations are performed:

1. The three distances,

   "+" *to* "-"
   (3 *to* **areg)** *min* **(3** *to* **breg)**
   and
   (fblatch *to* **areg)** *min* **(fblatch** *to* **breg)**

   which are 2, 5, and 9, respectively, resulting in a sum of 16.

2. Next an attempt is made to use "-" in the second operand as an identity operator. Its *identity cost* (6) is added to the distance from

   (+ 3 fblatch) *to* **areg**

   which is 23, resulting in a total of 29.

3. The same is also attempted with the other operand:

   (+ 3 fblatch) *to* **breg**

   resulting in a distance of 30, and a sum of 36.

4. The associative distance is 25. As in the previous case, there are no *difficult* operators, the max/min distance is 9, and the fixed constant is 10. Here, however the distance from "+" to "-" (2) and a size difference penalty of 4 are also added.

The distance function result is 16, the minimum of the first three computations; this is also the final result because the associative distance is larger and the size-based distance (4) is smaller.

One might think it peculiar that the

(+ 3 fblatch) *to* **(- areg breg)**

distance is smaller than that of

(+ 3 fblatch) *to* **(+ areg breg)**

since the expression pairs are identical except that the former has more distant operators. This anomaly is discussed in Section B.5.

The next example,

(+ 3 fblatch) *to* **areg**

was a subcomputation in the previous two examples:

1. The first lower bound for the distance function is the maximum "distance plus identity depth" from "3" or *fblatch* to *areg*. The distances are 15 and 19 respectively, and both identity depths are 4, so the result of this step is 23.
2. The second lower bound is the distance from "+" to *areg* (4) plus the smallest distance from the expression to any member of the set MuopExprs("+"). The two members of this set are

   (+ areg breg) *and* **(- areg breg)**

   In the previous examples, we saw that second of these expression gives us the smallest result, 16, so the value computed by this step is 22.
3. The associative distance is the sum of the distance from "+" to *areg* (6), the maximum distance of "3" or *fblatch* to areg (19), the size-based distance (4), and the fixed constant (10), or 39.

The largest of the first two results, 23, is selected as the *distance function* value; because the associative distance is larger, and the size difference (4) is larger, 23 is selected as the final result.

The attentive reader may have noticed that the evaluation of the distance from

(+ 3 fblatch) *to* **areg**

requires the evaluation of the distance from

(+ 3 fblatch) *to* **(+ areg breg)**

and vice versa, because the former evaluation computes the distance from

(+ 3 fblatch)

to each element of MuopExprs("+"). The caching mechanism ensures that indefinite recursion does not occur by prohibiting any computation to be performed when an identical computation is in progress.

The final example involves a resource with an index, estimating the distance from

(<- gpr[3] areg) *to* **(<- fbus (and areg breg))**

This distance is computed by adding the distances

(and areg breg) *to* **areg**
and
fbus *to* **gpr[3]**

The first of these is computed by adding the identity cost of the *AND* operation (4) to the smallest distance from any suboperand of the expression to *areg*, which in this case is 0. The second value is computed by adding table distance from *fbus to gpr* (2), to the *index cost* (0) which is the distance from "3" to **%wild**. Thus the total value is 6.

B.5. Shortcomings of the Evaluation Function

Although we have found that the evaluation function is usually effective in guiding the heuristic search, it should be evident from the examples that it computes only a rough approximation of the true cost of performing the actual transformation. The next few paragraphs discuss some of its weaknesses that became evident during experimentation.

A weakness mentioned previously is that the distance from

(+ 3 fblatch) *to* **(+ areg breg)**

was estimated to be greater than the distance from

(+ 3 fblatch) *to* **(- areg breg)**

This is because the evaluation function requires the operands to be matched in a one-to-one correspondence when the outermost operators are identical—thus either "3" or *fblatch* must be matched with *areg*—while the constraints are less strict when the operators are not identical—both "3" and *fblatch* can be match with *breg*. A one-to-one correspondence is not always possible when the outermost operators are different; the expressions may differ in the number of suboperands, for example. Thus the estimate may be less accurate, and sometimes lower, when the primary operators differ.

The evaluation function performs very poorly in the presence of expressions that include rotation or bit extraction operators. Our heuristic searches, for example, are not able to discover that a rotation by 8 can be performed by rotating by 5, and then later rotating by 3. It appears to us that in order to handle rotation and bit extraction correctly, it would be necessary to have b^2 separate distance tables for every one that currently exists—where b is the word length of the machine—in order to make estimates such as "the distance from resource *A*, rotated by 7, field length 5". During the course of this research, we attempted to approximate this information by adding 5 or 6 more tables, but the experiment was not successful.

Another inaccuracy in the evaluation function is its use of the *axiom factor* and multiples

thereof, to estimate the cost of unknown operations. In some cases the estimate is too high, while in others it is too low.

The size-based distance can also be a cause of inaccuracy because it assumes that the distance between two operands that differ greatly in size will be great. This is not true if there is an inexpensive μOp whose semantics are specified by a large expression. For example, if the μOp

```
(<- areg (and (+ %mask gpr[3]) (rot %wild fbus)))
```

had a cost of 2, the size-based distance would cause the total distance from

`(and (+ %mask gpr[3]) (rot %wild fbus))` *to* `areg`

to be 14 (7 times the axiom factor), even though the transformation could be performed by the search at a cost of 2.

Because the evaluation function is often inaccurate, one might ask the question, *Why not improve it?* We answer this by saying that we have improved it many times already—the reader only need refer to the Section B.3 to verify that it is quite complex; it is necessary to choose some stopping point in order to report on this research. The evaluation function appears to be accurate enough to be able to guide a large number of relatively deep searches.

Appendix C
List of Axioms Used in Experiments

This appendix contains the list of axioms that were used for the examples in chapters 6 and 8.

```
notid $1 :: (not (not $1));
anddemorg (and $1 $2) :: (not (or (eval (not $1)) (eval (not $2))));
ordemorg (or $1 $2) :: (not (and (eval (not $1)) (eval (not $2))));
unmindef (-- $1) :: (- 0 $1);
mindef (- $1 $2) :: (+ (+ $1 (eval (not $2))) 1);
plusid $1 :: (+ 0 $1);
unminid $1 :: (-- (eval (-- $1)));
unminnot (+ $1 -1) :: (not (eval (-- $1)));
plusnot (+ $2 (not $1)) :: (not (eval (- $1 $2)));
minnot (- $1 $2) :: (not (eval (+ (not $1) $2)));
minplusxfm (- $1 $2) :: (+ $1 (eval (-- $2)));
pluscommut (+ $1 $2) :: (+ $2 $1);
plusassoc (+ (+ $1 $2) $3) :: (+ $1 (eval (+ $2 $3)));
plusassoc2 (+ $1 (+ $2 $3)) :: (+ (eval (+ $1 $2)) $3);
orid $1 :: (or 0 $1);
andid $1 :: (and -1 $1);
xorid $1 :: (xor 0 $1);
andcommut (and $1 $2) :: (and $2 $1);
orcommut (or $1 $2) :: (or $2 $1);
xorcommut (xor $1 $2) :: (xor $2 $1);
andassoc (and (and $1 $2) $3) :: (and $1 (eval (and $2 $3)));
andassoc2 (and $1 (and $2 $3)) :: (and (eval (and $1 $2)) $3);
orassoc (or (or $1 $2) $3) :: (or $1 (eval (or $2 $3)));
orassoc2 (or $1 (or $2 $3)) :: (or (eval (or $1 $2)) $3);
xorassoc (xor (xor $1 $2) $3) :: (xor $1 (eval (xor $2 $3)));
xorassoc2 (xor $1 (xor $2 $3)) :: (xor (eval (xor $1 $2)) $3);
rotid $1 :: (rot 0 $1);
gtrminxfm (> $1 $2) :: (c3 0 $1 (not $2));
lssminxfm (< $1 $2) :: (c3 0 (not $1) $2);
lsszeroxfm (< $1 0) :: (rot 15 $1);
geqlssxfm (>= $1 $2) :: (not (< $1 $2));
leqgtrxfm (<= $1 $2) :: (not (> $1 $2));
eqlplusdef (= $1 $2) :: (bitand (+ (eval (not $1)) $2));
eqlxordef (= $1 $2) :: (bitand (xor (eval (not $1)) $2));
eqlonesdef (= $1 -1) :: (bitand $1);
zeroand 0 :: (and 0 ???{0 1});
onesor -1 :: (or -1 ???{0 1});
p3cary1id $1 :: (+ (eval (+ -1 $1)) 1);
concat2id $1 :: (@2 8 (rot 8 $1) $1);
concat3id $1 :: (@3 4 4 (rot 8 $1) (rot 4 $1) $1);
concat4id $1 :: (@4 1 2 12 (rot 15 $1) (rot 14 $1) (rot 12 $1) $1);
concat5id $1 :: (@5 1 1 1 12 (rot 15 $1) (rot 14 $1) (rot 13 $1)
                                        (rot 12 $1) $1);
subconcat (@2 $1 ???{0 1} $2) :: $2;
andgarbhi (@2 $1 0 $2) :: (and (eval (himask $1))
                               (@2 $1 ???{0 1} $2));
andgarblo (@2 $1 $2 0) :: (and (eval (lowmask (- 16 $1)))
                               (@2 $1 $2 ???{0 1}));
```

```
orgarbhi (@2 $1 -1 $2) :: (or (eval (lowmask (- 16 $1)))
                              (@2 $1 ???{0 1} $2));
orgarblo (@2 $1 $2 -1) :: (or (eval (himask $1))
                              (@2 $1 $3 ???{0 1}));
xorcom (not $1) :: (xor -1 $1);
eqlcommut (= $1 $2) :: (= $2 $1);
repack $1 :: (pack (mant $1) (expo $1));
expozero 0 :: (expo 0);
packzero 0 :: (pack 0 ???);
offflow (flow $1) :: $1;
notflow (flow (not $1)) :: (flow $1);
```

Appendix D
Kmap Machine Description

This appendix contains the machine description of the Kmap micromachine [Ousterhout 78] that was used in many of the examples. A sketch of the machine is given in Figure D-1.

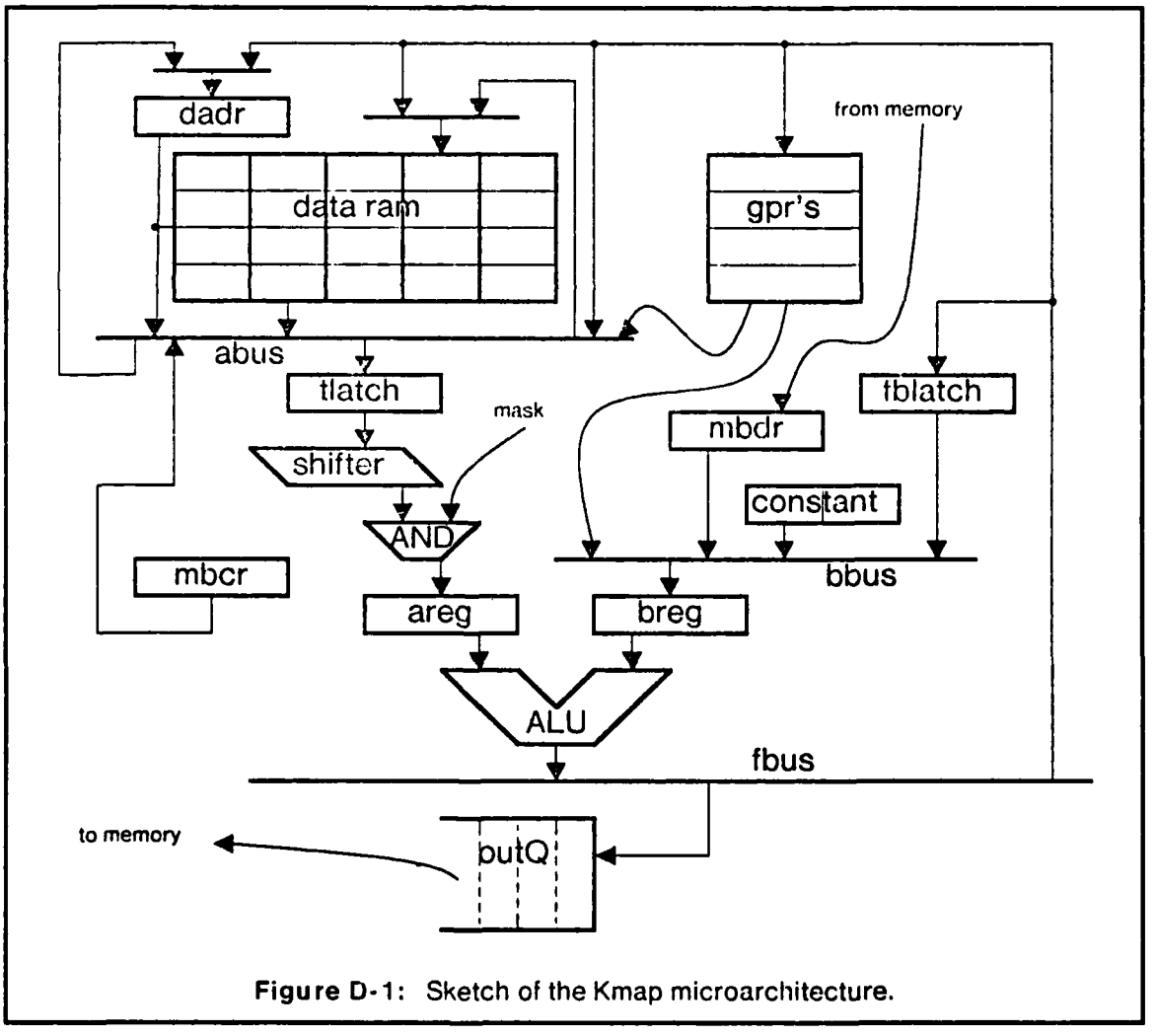

Figure D-1: Sketch of the Kmap microarchitecture.

The description is contained in three files. The first contains the names of all storage

resources in the micromachine. An asterisk (*) after a resource specifies that it is a *permanent* resource—that is that it may not be used to store temporary results. The numbers parentheses specify the word size and rank respectively.

```
madr * (12 0)  cxreg (8 0)  lincwd * (16 0)
fbus (16 0)  abus (16 0)  breg (16 0)  areg (16 0)
gpr (16 1)  dadr (12 0)  dram * (16 2)
gpridx (5 0)  tlatch (16 0)  scount (4 0)
conhi (8 0)  conlo (8 0)  cc1 (1 0)  cc2 (1 0)  cc16 (4 0)  carry (1 0)
carryin (1 0)  fblatch (16 0)  mbdr (16 0)  mbcr (16 0)
timeout (1 0)  refctl (6 0)  flaga (4 0)  flagb (4 0) dmask (16 0)
```

The second file contains the names of all conflict classes, each followed by its cost.

```
fbus 3  gpr 6  eop1 2  eop2 2
shift 2  areg 2  tlatch 1  gpridx 0
cc2 2  cc2s 1  breg 2  cc1 2
cc1s 1  fbl 1  flags 2  carry 2
dadr 2  abus 3  carryout 0  carryout1 0
carryout2 0  carryout3 0
```

The third file contains the μOp definitions.

```
nop {}
    (<- ???{0 1} ???{0 1})
constpinc {}
    (<- ???{0 1} ???{0 1})
shift {shift}
    (<- scount{4 9} %wild)
shift.fbus {shift}
    (<- scount{4 9} fbus{3 4})
areg.mask {areg}
    (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
ld.tl {tlatch}
    (<- tlatch{6 *} abus{5 6})
fbus.add {fbus carryout2 carryout3}
    (<- fbus{2 11} (+ (+ areg{0 1} breg{0 1}) carryin{0 1}))
carry.add {carryout carryout1}
    (<- carry{2 11} (c3 carryin{0 1} areg{0 1} breg{0 1}))
fbus.amb {fbus carryout1 carryout3}
    (<- fbus{2 11} (+ (+ areg{0 1} (not breg{0 1})) carryin{0 1}))
carry.amb {carryout carryout2}
    (<- carry{2 11} (c3 carryin{0 1} areg{0 1} (not breg{0 1})))
fbus.bma {fbus carryout1 carryout2}
    (<- fbus{2 11} (+ (+ (not areg{0 1}) breg{0 1}) carryin{0 1}))
carry.bma {carryout carryout3}
    (<- carry{2 11} (c3 carryin{0 1} (not areg{0 1}) breg{0 1}))
fbus.and {fbus carryout1 carryout2 carryout3}
    (<- fbus{2 11} (and areg{0 1} breg{0 1}))
fbus.or {fbus carryout1 carryout2 carryout3}
    (<- fbus{2 11} (or areg{0 1} breg{0 1}))
fbus.xor {fbus carryout1 carryout2 carryout3}
    (<- fbus{2 11} (xor areg{0 1} breg{0 1}))
fbus.zero {fbus carryout1 carryout2 carryout3}
    (<- fbus{2 11} 0)
fbus.ones {fbus carryout1 carryout2 carryout3}
    (<- fbus{2 11} -1)
ld.gpr {gpr}
    (<- gpr{8 *}[gpridx{2 3}] fbus{4 5})
gpridx {gpridx}
    (<- gpridx{0 9} %wild)
cc2.0 {cc2}
    (<- cc2{1 9} 0)
```

```
cc2.1 {cc2}
    (<- cc2{1 9} 1)
cc2.feven {cc2}
    (<- cc2{1 9} (not fbus{0 1}))
cc2.czero {cc2}
    (<- cc2{1 9} (not carry{0 1}))
cc2.fones {cc2}
    (<- cc2{1 9} (bitand fbus{0 1}))
breg.gpr {breg gpr}
   (<- breg{4 13} gpr{5 6}[gpridx{2 3}])
breg.fbl {breg}
    (<- breg{4 13} fblatch{3 4})
breg.con {breg}
    (<- breg{4 13} (@2 8 conhi{3 4} conlo{3 4}))
breg.mbdr {breg}
    (<- breg{4 13} mbdr{3 4})
breg.mbdlo {breg}
    (<- breg{4 13} (and 07777 mbdr{3 4}))
breg.ones {breg}
    (<- breg{4 13} -1)
cc1.0 {cc1}
    (<- cc1{1 9} 0)
cc1.1 {cc1}
    (<- cc1{1 9} 1)
cc1.fbus15 {cc1}
    (<- cc1{1 9} (rot 15 fbus{0 1}))
cc1.abus15 {cc1}
    (<- cc1{1 9} (rot 15 abus{0 1}))
cc1.abus14 {cc1}
    (<- cc1{1 9} (rot 14 abus{0 1}))
cc1.breg15 {cc1}
    (<- cc1{1 9} (rot 15 breg{0 1}))
cc1.timout {cc1}
    (<- cc1{1 9} timeout{0 1})
cc16.refctl {cc1}
    (<- cc16{1 9} refctl{0 1})
cc16.ahi {cc1}
    (<- cc16{1 9} (rot 12 abus{0 1}))
cc16.blo {cc1}
    (<- cc16{1 9} breg{0 1})
cc16.flo {cc1}
    (<- cc16{1 9} fbus{0 1})
cc16.flagb {cc1}
    (<- cc16{1 9} flagb{0 1})
cc16.flaga {cc1}
    (<- cc16{1 9} flaga{0 1})
ld.fbl {fbl}
    (<- fblatch{3 *} fbus{2 3})
ld.flaga {flags}
    (<- flaga{4 *} fbus{0 1})
ld.flagb {flags}
    (<- flagb{4 *} (@4 1 1 1 1 carry{0 1} cc2{3 4} cc1{3 4}))
carry.0 {carry}
    (<- carryin{0 9} 0)
carry.1 {carry}
    (<- carryin{0 9} 1)
carry.old {carry}
    (<- carryin{1 9} carry{0 1})
ld.conhi {eop1 eop2}
    (<- conhi{0 *} %wild)
ld.conlo {eop1 eop2}
    (<- conlo{0 *} %wild)
ld.d.fbus {eop1}
    (<- dram{8 *}[dadr{2 3} %wild] fbus{7 8})
ld.dr.aset {eop1}
    (<- dram{3 *}[dadr{2 3} %wild] (or dmask{1 2} abus{0 1}))
ld.dr.aclr {eop1}
    (<- dram{3 *}[dadr{2 3} %wild] (and (not dmask{1 2}) abus{0 1}))
ld.dmask {}
    (<- dmask{0 7} %bitset)
ld.dadr.a {dadr}
    (<- dadr{1 *} abus{0 1})
```

```
ld.dadr.f {dadr}
    (<- dadr{1 *} fbus{0 1})
abus.gpr {abus}
    (<- abus{5 12} gpr{2 3}[gpridx{2 3}])
abus.fbus {abus}
    (<- abus{5 12} fbus{2 3})
abus.edadr {abus}
    (<- abus{5 12} (@2 12 (hizero fbus{2 3}) dadr{2 3}))
abus.mbcr {abus}
    (<- abus{5 12} mbcr{0 1})
abus.pmcc {abus}
    (<- abus{5 12} (@5 1 1 1 12 0 carry{0 1} cc2{2 3} cc1{2 3} madr{0 1}))
abus.cxfl {abus}
    (<- abus{5 12} (@3 4 4 cxreg{0 1} flagb{4 5} flaga{4 5}))
abus.dram {abus}
    (<- abus{5 12} dram{4 5}[dadr{4 5} %wild])
abus.linc {abus eop1 eop2}
    (<- abus{5 12} lincwd{4 5})
br.cc1 {cc1s}
    (<- madr{6 15} (flow cc1{5 6}))
br.cc2 {cc2s}
    (<- madr{6 15} (flow cc2{5 6}))
br.cc16 {cc1s}
    (<- madr{6 15} (flow cc16{5 6}))
```

Appendix E
Puma Machine Description

This appendix contains the machine description a subset of the Puma micromachine [Grishman 78] that was used in our experiments. This model is inconsistent with the real machine in several respects. First, because our implementation assumed a maximum 16-bit word size (for the purposes of constant-folding, etc.) we also assume a maximum 16-bit word size, although the real machine has registers as wide as 60 bits. Secondly, many of the "exotic" μOps for setting condition codes have been omitted. Thirdly, although the ALU in the real machine is capable of both twos-complement and ones-complement arithmetic, our implementation is only capable of handling the former; μOps that perform ones-complement arithmetic are therefore omitted. A sketch of the microarchitecture is given in Figure E-1.

The description is contained in three files. The first contains the names of all storage resources in the micromachine. An asterisk (*) after a resource specifies that it is a *permanent* resource—that is that it may not be used to store temporary results. The numbers parentheses specify the word size and rank respectively.

```
mar * (10 0)  cond (1 0)  jfield (3 0)  kfield (3 0)
ifield (3 0)  ealu (12 0)  mq (16 0)  regoutput (16 0)
buffer (16 0)  alu (16 0)  carryin (1 0)  ac (16 0)
ilatch (3 0)  areg * (16 1)  breg * (16 1)  xreg * (16 1)
yreg (16 1)  reginput (16 0)  regidx (16 0)  e0 (12 0)
e1 (12 0)  e2 (12 0)  alux (16 0)  shiftlo (16 0)
cmrd (16 0)  preg (16 0)  shifthi (16 0)  ea (12 0)
eb (12 0)  mbus (16 0)  ma (16 0)  mem * (16 1)
```

The second file contains the names of all conflict classes, together with the cost assigned to each.

```
cc 2  reginput 0  reg 3  ridx 0
ilatch 1  buf 1  nomant 0  noexpo 0
noexpo2 0  mant 0  alu 2  carry 0
alux 0  shlo 0  shhi 0  ac 5
acl 0  ach 0  mq 2  lit75 5
lit73 5  lit79 5  ea 1  eb 1
ealu 3  preg 1  ma 1  io 1
```

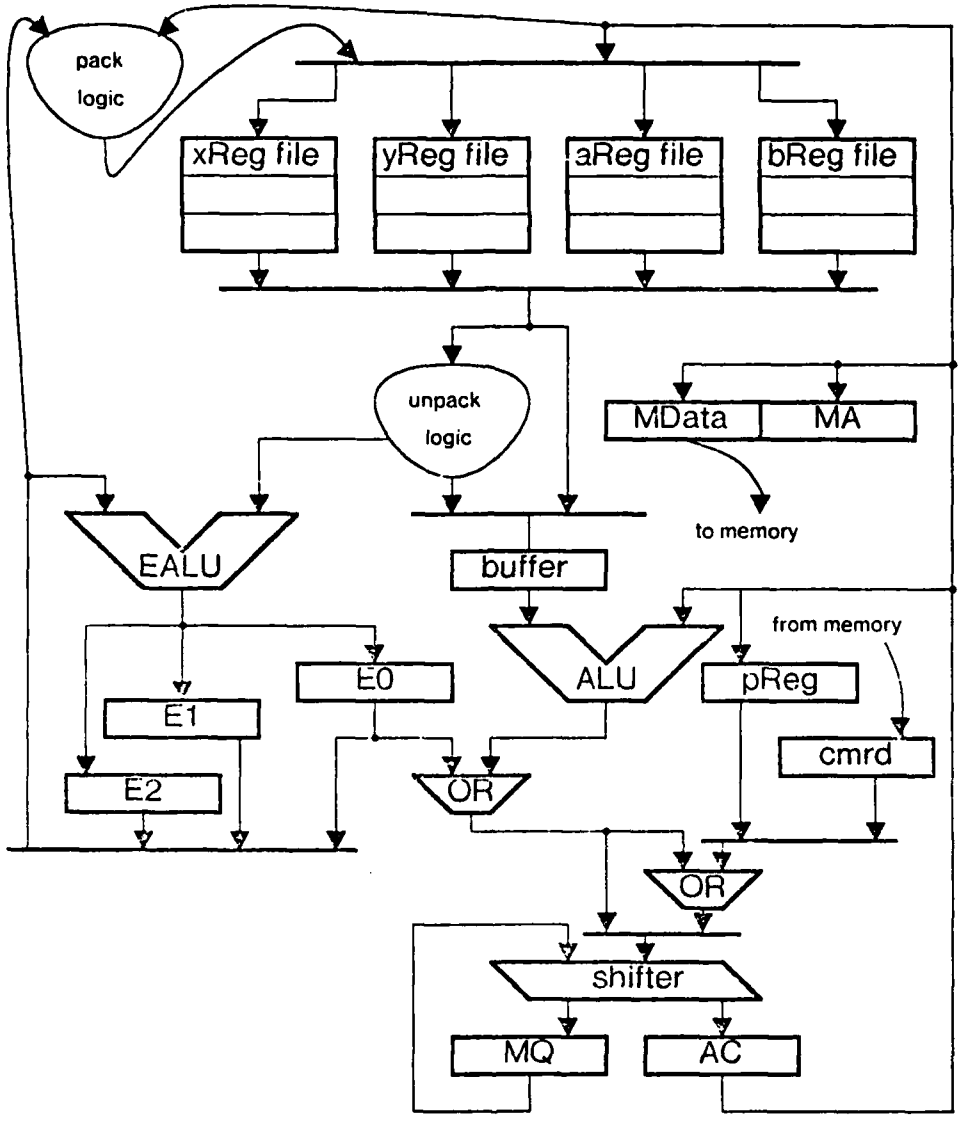

Figure E-1: Sketch of the Puma microarchitecture.

The third file contains the μOp definitions.

```
nop {}
 (<- ??? ???)
const.bind {}
 (<- ??? ???)
branch {}
 (<- mar{9 18} (flow cond{8 9}))
cc.j.0 {cc}
 (<- cond{8 9} (= jfield{0 1} 0))
cc.ealu.11 {cc}
 (<- cond{8 9} (rot 11 ealu{8 9}))
cc.ealu.4z {cc}
 (<- cond{8 9} (= 0 (and 017 ealu{8 9})))
cc.mq.4g7 {cc}
 (<- cond{8 9} (> (and 017 mq{8 9}) 7))
cc.mq.4g8 {cc}
 (<- cond{8 9} (> (and 017 mq{8 9}) 8))
cc.reg.15 {cc}
 (<- cond{8 9} (rot 15 regoutput{8 9}))
cc.buf.15 {cc}
 (<- cond{8 9} (rot 15 buffer{8 9}))
cc.alu.15 {cc}
 (<- cond{8 9} (rot 15 alu{8 9}))
cc.ac.15 {cc}
 (<- cond{8 9} (rot 15 ac{8 9}))
cc.il.0 {cc}
 (<- cond{8 9} ilatch{8 9})
cc.il.1 {cc}
 (<- cond{8 9} (rot 1 ilatch{8 9}))
cc.il.2 {cc}
 (<- cond{8 9} (rot 2 ilatch{8 9}))
cc.il.z {cc}
 (<- cond{8 9} (= 0 ilatch{8 9}))
cc.j.z {cc}
 (<- cond{8 9} (= 0 jfield{8 9}))
ld.areg {reg}
 (<- areg{4 *}[regidx{3 4}] reginput{3 4})
ld.breg {reg}
 (<- breg{4 *}[regidx{3 4}] reginput{3 4})
ld.xreg {reg}
 (<- xreg{4 *}[regidx{3 4}] reginput{3 4})
ld.yreg {reg}
 (<- yreg{4 *}[regidx{3 4}] reginput{3 4})
rd.areg {reg}
 (<- regoutput{2 11} areg{0 1}[regidx{1 2}])
rd.breg {reg}
 (<- regoutput{2 11} breg{0 1}[regidx{1 2}])
rd.xreg {reg}
 (<- regoutput{2 11} xreg{0 1}[regidx{1 2}])
rd.yreg {reg}
 (<- regoutput{2 11} yreg{0 1}[regidx{1 2}])
ridx.con {ridx}
 (<- regidx{0 9} %wild)
ridx.j {ridx}
 (<- regidx{0 9} jfield{0 1})
ridx.k {ridx}
 (<- regidx{0 9} kfield{0 1})
ridx.il {ridx}
 (<- regidx{0 9} ilatch{0 1})
ridx.mq {ridx}
 (<- regidx{0 9} mq{0 1})
ld.il {ilatch}
 (<- ilatch{4 *} ifield{0 1})
buf.reg {buf nomant noexpo}
 (<- buffer{4 *} regoutput{2 3})
buf.mant {buf mant}
 (<- buffer{4 *} (mant regoutput{2 3}))
reg.ac {reginput mant noexpo2}
 (<- reginput{1 10} ac{0 1})
```

```
reg.pack {reginput nomant}
 (<- reginput{1 10} (pack ac{0 1} e0{0 1}))
alu.0 {alu}
 (<- alu{2 11} 0)
alu.ones {alu}
 (<- alu{2 11} -1)
alu.ac {alu}
 (<- alu{2 11} (+ ac{0 1} carryin{0 1}))
alu.buf {alu}
 (<- alu{2 11} (+ buffer{0 1} carryin{0 1}))
alu.ng.ac {alu}
 (<- alu{2 11} (+ (not ac{0 1}) carryin{0 1}))
alu.ng.buf {alu}
 (<- alu{2 11} (+ (not buffer{0 1}) carryin{0 1}))
alu.plus {alu}
 (<- alu{2 11} (+ (+ ac{0 1} buffer{0 1}) carryin{0 1}))
alu.minus {alu}
 (<- alu{2 11} (+ (+ ac{0 1} (not buffer{0 1})) carryin{0 1}))
alu.or {alu}
 (<- alu{2 11} (or ac{0 1} buffer{0 1}))
alu.xor {alu}
 (<- alu{2 11} (xor ac{0 1} buffer{0 1}))
alu.and {alu}
 (<- alu{2 11} (and ac{0 1} buffer{0 1}))
alu.andnot {alu}
 (<- alu{2 11} (and ac{0 1} (not buffer{0 1})))
carry.0 {carry}
 (<- carryin 0)
carry.1 {carry}
 (<- carryin 1)
alux.alu {alux}
 (<- alux{2 11} alu{2 3})
alux.or {alux}
 (<- alux{2 11} (or alu{2 3} e0{0 1}))
shlo.pass {shlo}
 (<- shiftlo{2 11} alux{2 3})
shlo.cmrd {shlo}
 (<- shiftlo{2 11} (or alux{2 3} cmrd{0 1}))
shlo.k {shlo}
 (<- shiftlo{2 11} (or alux{2 3} kfield{0 1}))
shlo.preg {shlo}
 (<- shiftlo{2 11} (or alux{2 3} preg{0 1}))
shhi.mq {shhi}
 (<- shifthi{2 11} mq{0 1})
mq.hi {mq ach}
 (<- mq{4 *} shifthi{2 1})
mq.lo {mq acl}
 (<- mq{4 *} shiftlo{2 3})
ac.lo {ac acl}
 (<- ac{4 *} shiftlo{2 3})
ac.hi {ac ach}
 (<- ac{4 *} shifthi{2 3})
mq.0 {mq acl}
 (<- mq{4 *} 0)
mq.ones {mq acl}
 (<- mq{4 *} -1)
ea.con {ea lit75 lit73 lit79}
 (<- ea{2 11} %wild)
ea.e0 {ea}
 (<- ea{2 11} e0{0 1})
ea.e1 {ea}
 (<- ea{2 11} e1{0 1})
ea.e2 {ea}
 (<- ea{2 11} e2{0 1})
eb.ones {eb}
 (<- eb{2 11} -1)
eb.e0 {eb}
 (<- eb{2 11} e0{0 1})
eb.e1 {eb}
 (<- eb{2 11} e1{0 1})
eb.e2 {eb}
 (<- eb{2 11} e2{0 1})
```

```
eb.jk {eb}
 (<- eb{2 11} (@2 8 jfield{0 1} kfield{0 1}))
ealu.plus {ealu}
 (<- ealu{2 11} (+ ea{2 3} eb{2 3}))
ealu.minus {ealu}
 (<- ealu{2 11} (- ea{2 3} eb{2 3}))
ealu.expo {ealu noexpo noexpo2}
 (<- ealu{2 11} (expo regoutput{2 3}))
ld.e0 {}
 (<- e0{4 *} ealu{2 3})
ld.e1 {}
 (<- e1{4 *} ealu{2 3})
ld.e2 {}
 (<- e2{4 *} ealu{2 3})
ld.preg {preg lit75}
 (<- preg{4 *} ac{0 1})
inc.preg {preg lit75}
 (<- preg{4 *} (+ preg{0 1} 1))
dec.preg {preg lit75}
 (<- preg{4 *} (- preg{0 1} 1))
ma.preg {ma lit73}
 (<- ma{4 *} preg{0 1})
ma.ac {ma lit73}
 (<- ma{4 *} ac{0 1})
write.init {io lit79}
 (<- mbus{6 15} ac{5 6})
write.cont {io ac cc}
 (<- mem{4 *}[ma{0 1}] mbus{0 1})
read.init {io lit79}
 (<- mbus{7 16} mem{4 5}[ma{6 7}])
read.cont {io cc}
 (<- cmrd{4 *} mbus{0 1})
```

Appendix F
Selected Examples

This appendix contains three examples of the code generator and compaction routines in action. The first is a complete trace of the Puma example described in Appendix A, which discovers a code sequence that adds 5 to the *buffer* and stores the result in the *AC*. The other two examples are for the Kmap micromachine; the first uses the *squeeze* strategy to put the constant "−2" onto the *fbus*, while the third uses a combination of *And/Or* and *iteration* to move *lincwd* to a location in the *dram* and to move the value 7 onto the *fbus*.

The integers in braces denote the timing information as described in Chapter 5. Resource names without timing information are assumed by this implementation to have a timing value of **{0 1}**.

The timings listed after the heuristic searches in this section are not particularly accurate because the runs were made at time when the system was moderately loaded; paging and other overhead is included in the times listed.

```
search( 69.60): (<- ac (+ 0000005 buffer))
 ac.lo( 58.00)ac.hi( 60.00)
 feasible: ac.lo = (<- ac{4 9999} shiftlo{2 3})
  transform( 64.60): (+ 0000005 buffer) => shiftlo{2 3}
   applying fetch decomposition
    search( 64.60): (<- shiftlo{2 3} (+ 0000005 buffer))
    |shlo.pass( 53.00)shlo.cmrd( 59.03)shlo.k( 59.03)shlo.preg( 59.03)
    |feasible: shlo.pass = (<- shiftlo{2 11} alux{2 3})
     transform( 64.60): (+ 0000005 buffer) => alux{2 3}
      applying fetch decomposition
       search( 64.60): (<- alux{2 3} (+ 0000005 buffer))
        alux.alu( 62.00)alux.or( 50.13)
        feasible: alux.or = (<- alux{2 11} (or alu{2 3} e0))
        |transform( 64.60): (+ 0000005 buffer) => (or alu{2 3} e0)
         orid( 56.00)con-unfold( 58.70)con-unfold( 58.76)
         applying orid: $1 :: (or 0000000 $1) to (+ 0000005 buffer)
          transform( 64.60): (or 0000000 (+ 0000005 buffer)) => (or alu{2 3} e0)
           orcommut( 58.76)operandmatch( 56.00)
           decomposing by operand
            transform( 2.02): 0000000 => alu{2 3}
            |applying fetch decomposition
             search( 2.02): (<- alu{2 3} 0000000)
              alu.0( 2.00)
              feasible: alu.0 = (<- alu{2 11} 0000000)
               ... success on search( 2.02) with  2.00
            |... success on transform( 2.02) with  2.00
            transform( 62.58): (+ 0000005 buffer) => e0
            |applying fetch decomposition
             search( 62.58): (<- e0 (+ 0000005 buffer))
              ld.e0( 54.00)
              feasible: ld.e0 = (<- e0{4 9999} ealu{2 3})
               transform( 62.58): (+ 0000005 buffer) => ealu{2 3}
                applying fetch decomposition
                |search( 62.58): (<- ealu{2 3} (+ 0000005 buffer))
                 ealu.plus( 58.00)ealu.minus( 62.00)
                 feasible: ealu.plus = (<- ealu{2 11} (+ ea{2 3} eb{2 3}))
                  transform( 59.58): (+ 0000005 buffer) => (+ ea{2 3} eb{2 3})
                   pluscommut( 57.00)operandmatch( 55.00)
                   decomposing by operand
                    transform( 16.68): 0000005 => ea{2 3}
                    |applying fetch decomposition
```

```
search( 16.68): (<- ea{2 3} 0000006)
 ea.con( 16.00)
 feasible: ea.con = (<- ea{2 11} %wild)
  transform( 0.00): 0000006 => %wild
   attempting constant match
   |it's a match!!
   ... success on transform( 0.00) with  0.00
  ... success on search( 16.68) with  16.00
... success on transform( 16.68) with  16.00
transform( 42.90): buffer => eb{2 3}
|applying fetch decomposition
 search( 42.90): (<- eb{2 3} buffer)
  eb.e0( 39.00)eb.e1( 39.00)eb.e2( 39.00)
  feasible: eb.e0 = (<- eb{2 11} e0)
   transform( 41.90): buffer => e0
    applying fetch decomposition
    |search( 41.90): (<- e0 buffer)
     ld.e0( 38.00)
     feasible: ld.e0 = (<- e0{4 9999} ealu{2 3})
      transform( 41.90): buffer => ealu{2 3}
       applying fetch decomposition
        search( 41.90): (<- ealu{2 3} buffer)
        |ealu.expo( 38.00)
        |feasible: ealu.expo = (<- ealu{2 11} (expo regoutput{2 3}))
         transform( 38.90): buffer => (expo regoutput{2 3})
          No takers!
          ... cutoff reached.
           ... fail on transform( 38.90)
        |... cutoff reached.
        |... fail on search( 41.90)
       ... fail on transform( 41.90)
     ... cutoff reached.
     ... fail on search( 41.90)
    ... fail on transform( 41.90)
  feasible: eb.e1 = (<- eb{2 11} e1)
   transform( 41.90): buffer => e1
    applying fetch decomposition
    |search( 41.90): (<- e1 buffer)
    | No takers!
    | ... cutoff reached.
    | ... fail on search( 41.90)
    ... fail on transform( 41.90)
  feasible: eb.e2 = (<- eb{2 11} e2)
   transform( 41.90): buffer => e2
    applying fetch decomposition
    |search( 41.90): (<- e2 buffer)
    | No takers!
    | ... cutoff reached.
    | ... fail on search( 41.90)
    ... fail on transform( 41.90)
  ... cutoff reached.
  ... fail on search( 42.90)
 ... fail on transform( 42.90)
applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0000006 buffer)
transform( 59.58): (+ buffer 0000006) => (+ ea{2 3} eb{2 3})
|pluscommut( 55.00)
|applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ buffer 0000006)
| transform( 59.58): (+ 0000006 buffer) => (+ ea{2 3} eb{2 3})
|  ... found previous failure
|  ... fail on transform( 59.58)
|... cutoff reached.
|... fail on transform( 59.68)
... cutoff reached.
... fail on transform( 59.58)
feasible: ealu.minus = (<- ealu{2 11} (- ea{2 3} eb{2 3}))
 transform( 59.58): (+ 0000006 buffer) => (- ea{2 3} eb{2 3})
  No takers!
  ... cutoff reached.
  ... fail on transform( 59.58)
 ... cutoff reached.
 ... fail on search( 62.68)
... fail on transform( 62.68)
... cutoff reached.
... fail on search( 62.68)
... fail on transform( 62.68)
applying orcommut: (or $1 $2) :: (or $2 $1) to (or 0000000 (+ 0000006 buffer))
transform( 64.60): (or (+ 0000006 buffer) 0000000) => (or alu{2 3} e0)
|orcommut( 56.00)
|applying orcommut: (or $1 $2) :: (or $2 $1) to (or (+ 0000006 buffer) 0000000)
| transform( 64.60): (or 0000000 (+ 0000006 buffer)) => (or alu{2 3} e0)
|  ... found previous failure
|  ... fail on transform( 64.60)
|... cutoff reached.
|... fail on transform( 64.60)
... cutoff reached.
... fail on transform( 64.60)
applying con-unfold to (+ 0000006 buffer)
transform( 64.60): (+ (+ 0000006 0777777) buffer) => (or alu{2 3} e0)
 No takers!
 ... cutoff reached.
 ... fail on transform( 64.60)
applying con-unfold to (+ 0000006 buffer)
transform( 64.60): (+ (+ 0000004 0000001) buffer) => (or alu{2 3} e0)
 No takers!
 ... cutoff reached.
 ... fail on transform( 64.60)
... cutoff reached.
... fail on transform( 64.60)
feasible: alux.alu = (<- alux{2 11} alu{2 3})
transform( 64.60): (+ 0000006 buffer) => alu{2 3}
 applying fetch decomposition
  search( 64.60): (<- alu{2 3} (+ 0000006 buffer))
   alu.ng.buf( 63.00)alu.plus( 63.00)alu.minus( 57.00)alu.xor( 57.00)alu.andnot( 63.00)
   feasible: alu.plus = (<- alu{2 11} (+ (+ ac buffer) carryin))
```

```
transform( 62.60): (+ 0000006 buffer) => (+ (+ ac buffer) carryin)
plusid( 42.00)pluscommut( 51.00)con-unfold( 60.00)
applying plusid: $1 :: (+ 0000000 $1) to (+ 0000006 buffer)
 transform( 62.60): (+ 0000000 (+ 0000006 buffer)) => (+ (+ ac buffer) carryin)
  pluscommut( 42.00)plusassoc2( 61.00)
  applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0000000 (+ 0000006 buffer))
   transform( 62.60): (+ (+ 0000006 buffer) 0000000) => (+ (+ ac buffer) carryin)
    pluscommut( 42.00)plusassoc( 62.16)operandmatch( 42.00)
    decomposing by operand
     transform( 0.00): 0000000 => carryin
      applying fetch decomposition
       search( 0.00): (<- carryin 0000000)
        carry.0( 0.00)
        feasible: carry.0 = (<- carryin 0000000)
        ... success on search( 0.00) with  0.00
      ... success on transform( 0.00) with  0.00
     transform( 62.60): (+ 0000006 buffer) => (+ ac buffer)
      pluscommut( 42.00)con-unfold( 51.00)con-unfold( 61.00)operandmatch( 42.00)
      decomposing by operand
       transform( 62.60): 0000006 => ac
        applying fetch decomposition
         search( 62.60): (<- ac 0000006)
          ac.lo( 42.00)ac.hi( 44.00)
          feasible: ac.lo = (<- ac{4 9999} shiftlo{2 3})
           transform( 57.60): 0000005 => shiftlo{2 3}
            applying fetch decomposition
             search( 57.60): (<- shiftlo{2 3} 0000006)
              shlo.pass( 37.00)shlo.cmrd( 48.00)shlo.k( 48.00)shlo.preg( 48.00)
              feasible: shlo.pass = (<- shiftlo{2 11} alux{2 3})
               transform( 57.60): 0000006 => alux{2 3}
                applying fetch decomposition
                 search( 57.60): (<- alux{2 3} 0000006)
                  alux.alu( 53.00)alux.or( 37.00)
                  feasible: alux.or = (<- alux{2 11} (or alu{2 3} e0))
                   transform( 57.60): 0000005 => (or alu{2 3} e0)
                    orid( 30.00)
                    applying orid: $1 :: (or 0000000 $1) to 0000005
                     transform( 57.60): (or 0000000 0000005) => (or alu{2 3} e0)
                      orcommut( 49.06)orid( 56.26)operandmatch( 30.00)
                      decomposing by operand
                       transform( 2.20): 0000000 => alu{2 3}
                        |(using previous result)
                        ... success on transform( 2.20) with  2.00
                       transform( 55.40): 0000005 => e0
                        applying fetch decomposition
                         search( 55.40): (<- e0 0000005)
                          ld.e0( 28.00)
                          feasible: ld.e0 = (<- e0{4 9999} ealu{2 3})
                           transform( 56.40): 0000005 => ealu{2 3}
                            applying fetch decomposition
                             search( 55.40): (<- ealu{2 3} 0000005)
                             ealu.plus( 28.00)ealu.minus( 32.00)
                             feasible: ealu.plus = (<- ealu{2 11} (+ ea{2 3} eb{2 3}))
                              transform( 62.40): 0000005 => (+ ea{2 3} eb{2 3})
                               con-unfold( 17.00)p3carry1d( 34.00)
                               applying con-unfold to 0000005
                                transform( 52.40): (+ 0000006 0777777) => (+ ea{2 3} eb{2 3})
                                 con-unfold( 26.00)pluscommut( 34.00)operandmatch( 17.00)
                                 decomposing by operand
                                  transform( 1.26): 0777777 => eb{2 3}
                                   applying fetch decomposition
                                    search( 1.26): (<- eb{2 3} 0777777)
                                     eb.ones( 1.00)
                                     feasible: eb.ones = (<- eb{2 11} 0777777)
                                     ... success on search( 1.26) with  1.00
                                   ... success on transform( 1.26) with  1.00
                                  transform( 51.14): 0000006 => ea{2 3}
                                   applying fetch decomposition
                                    search( 51.14): (<- ea{2 3} 0000006)
                                     ea.con( 16.00)ea.e0( 29.00)ea.e1( 29.00)ea.e2( 29.00)
                                     feasible: ea.con = (<- ea{2 11} %wild)
                                      transform( 0.00): 0000006 => %wild
                                       |attempting constant match
                                       | it's a match!!
                                       |... success on transform( 0.00) with  0.00
                                      ... success on search( 51.14) with  16.00
                                    ... success on transform( 51.14) with  16.00
                                   ... success on transform( 52.40) with  17.00
                                  ... success on transform( 52.40) with  17.00
                                 ... success on search( 55.40) with  20.00
                                ... success on transform( 55.40) with  20.00
                               ... success on search( 56.40) with  20.00
                              ... success on transform( 55.40) with  20.00
                             ... success on transform( 57.60) with  22.00
                            ... success on transform( 57.60) with  22.00
                           ... success on search( 57.60) with  22.00
                          ... success on transform( 57.60) with  22.00
                         ... success on search( 57.60) with  22.00
                        ... success on transform( 57.60) with  22.00
                       ... success on search( 62.60) with  27.00
                      ... success on transform( 62.60) with  27.00
                     ... success on transform( 62.60) with  27.00
                    ... success on transform( 62.60) with  27.00
                   ... success on transform( 62.60) with  27.00
                  ... success on transform( 62.60) with  27.00
                 ... success on search( 64.60) with  29.00
                ... success on transform( 64.60) with  29.00
               ... success on search( 64.60) with  29.00
              ... success on transform( 64.60) with  29.00
             ... success on search( 64.60) with  29.00
            ... success on transform( 64.60) with  29.00
           ... success on search( 69.60) with  34.00
63 nodes examined.
Maximum search depth: 28
```

```
Maximum axiom depth: 4
Approximate execution time: 48.33 seconds

Compacting:
ld.e0  eb.ones  ealu.plus  ea.con 0000006 (0)
ac.lo  shlo.pass  alux.or  alu.0  (1)
ac.lo  shlo.pass  alux.alu  alu.plus  carry.0  (2)
```

In this example, the search fails at first but succeeds on the second try.

```
search( 16.90): (<- fbus 0777776)
 fbus.and( 13.00)fbus.or( 13.00)fbus.xor( 13.00)
 feasible: fbus.and * (<- fbus{2 11} (and areg breg))
  transform( 13.90): 0777776 => (and areg breg)
   andid( 8.00)
   applying andid: $1 :: (and 0777777 $1) to 0777776
    transform( 13.90): (and 0777777 0777776) => (and areg breg)
    |andcommut( 8.00)andid( 13.00)
    |applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 0777776)
     transform( 13.90): (and 0777776 0777777) => (and areg breg)
      andcommut( 8.00)andid( 13.00)operandmatch( 8.00)
      decomposing by operand
       transform( 2.74): 0777777 => breg
        applying fetch decomposition
        |search( 2.74): (<- breg 0777777)
         breg.ones( 2.00)
         feasible: breg.ones * (<- breg{4 13} 0777777)
         ... success on search( 2.74) with  2.00
       ... success on transform( 2.74) with  2.00
      transform( 11.16): 0777776 => areg
       applying fetch decomposition
       |search( 11.16): (<- areg 0777776)
        areg.mask( 6.00)
        feasible: areg.mask * (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
         transform( 9.16): 0777776 => (and %mask (rot scount{7 8} tlatch{7 8}))
          andid( 9.00)
          applying andid: $1 :: (and 0777777 $1) to 0777776
           transform( 9.16): (and 0777777 0777776) => (and %mask (rot scount{7 8} tlatch{7 8}))
           |andcommut( 9.00)
           |applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 0777776)
            transform( 9.16): (and 0777776 0777777) => (and %mask (rot scount{7 8} tlatch{7 8}))
             andcommut( 9.00)operandmatch( 9.00)
             decomposing by operand
              transform( 0.00): 0777776 => %mask
               attempting constant match
               |it's a match!!
               ... success on transform( 0.00) with  0.00
              transform( 9.16): 0777777 => (rot scount{7 8} tlatch{7 8})
               rotid( 9.00)
               applying rotid: $1 :: (rot 0000000 $1) to 0777777
               |transform( 9.16): (rot 0000000 0777777) => (rot scount{7 8} tlatch{7 8})
                operandmatch( 9.00)
                decomposing by operand
                 transform( 2.02): 0000000 => scount{7 8}
                  applying fetch decomposition
                   search( 2.02): (<- scount{7 8} 0000000)
                   |shift( 2.00)
                   |feasible: shift * (<- scount{4 9} %wild)
                   | transform( 0.00): 0000000 => %wild
                   |  attempting constant match
                   |  it's a match!!
                   | ... success on transform( 0.00) with  0.00
                   |... success on search( 2.02) with  2.00
                  ... success on transform( 2.02) with  2.00
                 transform( 7.15): 0777777 => tlatch{7 8}
                  applying fetch decomposition
                   search( 7.15): (<- tlatch{7 8} 0777777)
                   |ld.tl( 7.00)
                   |feasible: ld.tl * (<- tlatch{6 9999} abus{5 6})
                   | transform( 6.15): 0777777 => abus{5 6}
                   |  applying fetch decomposition
                   |   search( 6.15): (<- abus{5 6} 0777777)
                   |    abus.fbus( 6.00)
                   |    feasible: abus.fbus * (<- abus{5 12} fbus{2 3})
                   |    |transform( 3.15): 0777777 => fbus{2 3}
                   |    | applying fetch decomposition
                   |    |  search( 3.15): (<- fbus{2 3} 0777777)
                   |    |   fbus.ones( 3.00)
                   |    |   feasible: fbus.ones * (<- fbus{2 11} 0777777)
                   |    |    ... squeezed out.
                   |    |   ... cutoff reached.
                   |    |   ... fail on search( 3.15)
                   |    |  ... fail on transform( 3.15)
                   |    | ... cutoff reached.
                   |    | ... fail on search( 6.15)
                   |    ... fail on transform( 6.15)
                   |   ... cutoff reached.
                   |   ... fail on search( 7.15)
                   |  ... fail on transform( 7.15)
                   | ... cutoff reached.
                   | ... fail on transform( 9.16)
                   ... cutoff reached.
                   ... fail on transform( 9.16)
                 applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777776 0777777)
                  transform( 9.16): (and 0777777 0777776) => (and %mask (rot scount{7 8} tlatch{7 8}))
                   ... found previous failure
                   ... fail on transform( 9.16)
                  ... cutoff reached.
                  ... fail on transform( 9.16)
                 ... cutoff reached.
```

```
|  |  |  |... fail on transform( 9.16)
|  |  |  ... cutoff reached.
|  |  |  ... fail on transform( 9.16)
|  |  | ... cutoff reached.
|  |  | ... fail on search( 11.18)
|  |  ... fail on transform( 11.16)
|  | applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777776 0777777)
|  |  transform( 13.90): (and 0777777 0777776) => (and areg breg)
|  |   ... found previous failure
|  |   ... fail on transform( 13.90)
|  | applying andid: $1 :: (and 0777777 $1) to (and 0777776 0777777)
|  |  transform( 13.90): (and 0777777 (and 0777776 0777777)) => (and areg breg)
|  |   andassoc2( 8.00)andcommut( 13.00)
|  |   applying andassoc2: (and $1 (and $2 $3)) :: (and (eval (and $1 $2)) $3) to (and 0777777 (and 0777776 0777777
|  |  |transform( 13.90): (and 0777776 0777777) => (and areg breg)
|  |  | ... found previous failure
|  |  | ... fail on transform( 13.90)
|  |  applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 (and 0777776 0777777))
|  |  |transform( 13.90): (and (and 0777776 0777777) 0777777) => (and areg breg)
|  |  | andassoc( 8.00)andcommut( 13.00)operandmatch( 13.00)
|  |  | applying andassoc: (and (and $1 $2) $3) :: (and $1 (eval (and $2 $3))) to (and (and 0777776 0777777) 07777
|  |  |  transform( 13.90): (and 0777776 0777777) => (and areg breg)
|  |  |   ... found previous failure
|  |  |   ... fail on transform( 13.90)
|  |  | decomposing by operand
|  |  |  transform( 2.04): 0777777 => breg
|  |  |    (using previous result)
|  |  |   ... success on transform( 2.04) with  2.00
|  |  |  transform( 11.86): (and 0777776 0777777) => areg
|  |  |   applying fetch decomposition
|  |  |    search( 11.86): (<- areg (and 0777776 0777777))
|  |  |    |No takers!
|  |  |    |... cutoff reached.
|  |  |    |... fail on search( 11.86)
|  |  |   ... fail on transform( 11.86)
|  |  | applying andcommut: (and $1 $2) :: (and $2 $1) to (and (and 0777776 0777777) 0777777)
|  |  |  transform( 13.90): (and 0777777 (and 0777776 0777777)) => (and areg breg)
|  |  |   ... found previous failure
|  |  |   ... fail on transform( 13.90)
|  |  | ... cutoff reached.
|  |  | ... fail on transform( 13.90)
|  |  ... cutoff reached.
|  |  ... fail on transform( 13.90)
|  | ... cutoff reached.
|  | ... fail on transform( 13.90)
| applying andid: $1 :: (and 0777777 $1) to (and 0777777 0777776)
|  transform( 13.90): (and 0777777 (and 0777777 0777776)) => (and areg breg)
|   andassoc2( 8.00)andcommut( 13.00)
|   applying andassoc2: (and $1 (and $2 $3)) :: (and (eval (and $1 $2)) $3) to (and 0777777 (and 0777777 0777776))
|    transform( 13.90): (and 0777777 0777776) => (and areg breg)
|     ... found previous failure
|     ... fail on transform( 13.90)
|   applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 (and 0777777 0777776))
|    transform( 13.90): (and (and 0777777 0777776) 0777777) => (and areg breg)
|     andassoc( 8.00)andcommut( 13.00)operandmatch( 13.00)
|     applying andassoc: (and (and $1 $2) $3) :: (and $1 (eval (and $2 $3))) to (and (and 0777777 0777776) 0777777
|     |transform( 13.90): (and 0777777 0777776) => (and areg breg)
|     | ... found previous failure
|     | ... fail on transform( 13.90)
|     decomposing by operand
|     |transform( 2.04): 0777777 => breg
|     |  (using previous result)
|     | ... success on transform( 2.04) with  2.00
|     |transform( 11.86): (and 0777777 0777776) => areg
|     | applying fetch decomposition
|     |  search( 11.86): (<- areg (and 0777777 0777776))
|     |   No takers!
|     |   ... cutoff reached.
|     |   ... fail on search( 11.86)
|     | ... fail on transform( 11.86)
|     applying andcommut: (and $1 $2) :: (and $2 $1) to (and (and 0777777 0777776) 0777777)
|     |transform( 13.90): (and 0777777 (and 0777777 0777776)) => (and areg breg)
|     |  ... found previous failure
|     | ... fail on transform( 13.90)
|     ... cutoff reached.
|     ... fail on transform( 13.90)
|    ... cutoff reached.
|    ... fail on transform( 13.90)
|   ... cutoff reached.
|   ... fail on transform( 13.90)
|  ... cutoff reached.
|  ... fail on transform( 13.90)
... cutoff reached.
... fail on transform( 13.90)
feasible: fbus.or = (<- fbus{2 11} (or areg breg))
 transform( 13.90): 0777776 => (or areg breg)
  No takers!
  ... cutoff reached.
  ... fail on transform( 13.90)
feasible: fbus.xor = (<- fbus{2 11} (xor areg breg))
 transform( 13.90): 0777776 => (xor areg breg)
  No takers!
  ... cutoff reached.
  ... fail on transform( 13.90)
... cutoff reached.
... fail on search( 16.80)
43 nodes examined.
Maximum search depth: 17
Maximum axiom depth: 6
Approximate execution time: 12.80 seconds

search( 21.97): (<- fbus 0777776)
 fbus.add( 21.00)fbus.bma( 21.00)fbus.and( 13.00)fbus.or( 13.00)fbus.xor( 13.00)
 feasible: fbus.and = (<- fbus{2 11} (and areg breg))
  transform( 18.97): 0777776 => (and areg breg)
   andid( 8.00)
```

```
applying andid: $1 :: (and 0777777 $1) to 0777776
 transform( 18.97): (and 0777777 0777776) => (and areg breg)
 |andcommut( 8.00)andid( 13.00)operandmatch( 18.00)
 |applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 0777776)
  transform( 18.37): (and 0777776 0777777) => (and areg breg)
   andcommut( 8.90)andid( 13.00)operandmatch( 8.00)
   decomposing by operand
    transform( 3.37): 0777777 => breg
     |(using previous result)
     ... success on transform( 3.37) with  2.00
    transform( 15.60): 0777776 => areg
     applying fetch decomposition
     |search( 15.60): (<- areg 0777776)
      areg.mask( 6.00)
      feasible: areg.mask • (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
       transform( 13.60): 0777776 => (and %mask (rot scount{7 8} tlatch{7 8}))
        andid( 9.00)
        applying andid: $1 :: (and 0777777 $1) to 0777776
         transform( 13.60): (and 0777777 0777776) => (and %mask (rot scount{7 8} tlatch{7 8}))
          |andcommut( 9.00)con-unfold( 10.23)
          |applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 0777776)
           transform( 13.60): (and 0777776 0777777) => (and %mask (rot scount{7 8} tlatch{7 8}))
            andcommut( 9.00)con-unfold( 10.23)operandmatch( 9.00)
            decomposing by operand
             transform( 0.00): 0777776 => %mask
              |(using previous result)
              ... success on transform( 0.00) with  0.00
             transform( 13.60): 0777777 => (rot scount{7 8} tlatch{7 8})
              rotid( 9.00)
              applying rotid: $1 :: (rot 0000000 $1) to 0777777
              |transform( 13.60): (rot 0000000 0777777) => (rot scount{7 8} tlatch{7 8})
              | operandmatch( 9.00)
              | decomposing by operand
              |  transform( 2.45): 0000000 => scount{7 8}
              |   (using previous result)
              |   ... success on transform( 2.45) with  2.00
              |  transform( 11.15): 0777777 => tlatch{7 8}
              |   applying fetch decomposition
              |   search( 11.15): (<- tlatch{7 8} 0777777)
              |   |ld.tl( 7.00)
              |   |feasible: ld.tl • (<- tlatch{6 9999} abus{5 6})
              |   | transform( 10.15): 0777777 => abus{5 6}
              |   |  applying fetch decomposition
              |   |  search( 10.15): (<- abus{5 6} 0777777)
              |   |   abus.fbus( 6.00)abus.dram( 8.00)
              |   |   feasible: abus.fbus • (<- abus{5 12} fbus{2 3})
              |   |   |transform( 7.15): 0777777 => fbus{2 3}
              |   |   | applying fetch decomposition
              |   |   | search( 7.15): (<- fbus{2 3} 0777777)
              |   |   |  fbus.ones( 3.00)
              |   |   |  feasible: fbus.ones • (<- fbus{2 11} 0777777)
              |   |   |   ... squeezed out.
              |   |   |  ... cutoff reached.
              |   |   |  ... fail on search( 7.15)
              |   |   | ... fail on transform( 7.15)
              |   |   feasible: abus.dram • (<- abus{5 12} dram{4 5}[dadr{4 5} %wild])
              |   |   |transform( 7.15): 0777777 => dram{4 5}[dadr{4 5} %wild]
              |   |   | applying fetch decomposition
              |   |   | search( 7.15): (<- dram{4 5}[dadr{4 5} %wild] 0777777)
              |   |   |  No takers!
              |   |   |  ... cutoff reached.
              |   |   |  ... fail on search( 7.15)
              |   |   | ... fail on transform( 7.15)
              |   |   ... cutoff reached.
              |   |   ... fail on search( 10.15)
              |   | ... fail on transform( 10.15)
              |   ... cutoff reached.
              |   ... fail on search( 11.15)
              | ... fail on transform( 11.15)
              ... cutoff reached.
              ... fail on transform( 13.60)
             ... cutoff reached.
             ... fail on transform( 13.60)
            applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777776 0777777)
             transform( 13.60): (and 0777777 0777776) => (and %mask (rot scount{7 8} tlatch{7 8}))
              ... found previous failure
              ... fail on transform( 13.60)
            applying con-unfold to (and 0777776 0777777)
             transform( 13.60): (and (rot 0000017 0077777) 0777777) => (and %mask (rot scount{7 8} tlatch{7 8}))
              andcommut( 10.23)
              applying andcommut: (and $1 $2) :: (and $2 $1) to (and (rot 0000017 0077777) 0777777)
              |transform( 13.60): (and 0777777 (rot 0000017 0077777)) => (and %mask (rot scount{7 8} tlatch{7 8}
              | andcommut( 10.23)operandmatch( 12.26)
              | applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 (rot 0000017 0077777))
              |  transform( 13.60): (and (rot 0000017 0077777) 0777777) => (and %mask (rot scount{7 8} tlatch{7
              |   ... found previous failure
              |   ... fail on transform( 13.60)
              |  decomposing by operand
              |   transform( 0.00): 0777777 => %mask
              |    attempting constant match
              |    it's a match!!
              |   ... success on transform( 0.00) with  0.00
              |  transform( 13.60): (rot 0000017 0077777) => (rot scount{7 8} tlatch{7 8})
              |   No takers!
              |   ... cutoff reached.
              |   ... fail on transform( 13.60)
              | ... cutoff reached.
              | ... fail on transform( 13.60)
              ... cutoff reached.
              ... fail on transform( 13.60)
             ... cutoff reached.
             ... fail on transform( 13.60)
            applying con-unfold to (and 0777777 0777776)
             transform( 13.60): (and 0777777 (rot 0000017 0077777)) => (and %mask (rot scount{7 8} tlatch{7 8}))
```

```
... found previous failure
... fail on transform( 13.60)
... cutoff reached.
... fail on transform( 13.60)
... cutoff reached.
... fail on transform( 13.60)
... cutoff reached.
... fail on search( 15.60)
... fail on transform( 16.60)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777776 0777777)
transform( 18.97): (and 0777777 0777776) => (and areg breg)
... found previous failure
... fail on transform( 18.97)
applying andid: $1 :: (and 0777777 $1) to (and 0777776 0777777)
transform( 18.97): (and 0777777 (and 0777776 0777777)) => (and areg breg)
andassoc2( 8.00)andcommut( 13.00)
applying andassoc2: (and $1 (and $2 $3)) :: (and (eval (and $1 $2)) $3) to (and 0777777 (and 0777776 0777777
transform( 18.97): (and 0777776 0777777) => (and areg breg)
... found previous failure
... fail on transform( 18.97)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 (and 0777776 0777777))
transform( 18.97): (and (and 0777776 0777777) 0777777) => (and areg breg)
andassoc( 8.00)andcommut( 13.00)operandmatch( 13.00)
applying andassoc: (and (and $1 $2) $3) :: (and $1 (eval (and $2 $3))) to (and (and 0777776 0777777) 07777
transform( 18.97): (and 0777776 0777777) => (and areg breg)
... found previous failure
... fail on transform( 18.97)
decomposing by operand
transform( 2.26): 0777777 => breg
(using previous result)
... success on transform( 2.26) with  2.00
transform( 16.71): (and 0777776 0777777) => areg
applying fetch decomposition
search( 16.71): (<- areg (and 0777776 0777777))
areg.mask( 11.00)
feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
transform( 14.71): (and 0777776 0777777) => (and %mask (rot scount{7 8} tlatch{7 8}))
andcommut( 9.00)con-unfold( 10.23)operandmatch( 9.00)
decomposing by operand
transform( 0.00): 0777776 => %mask
(using previous result)
... success on transform( 0.00) with  0.00
transform( 14.71): 0777777 => (rot scount{7 8} tlatch{7 8})
rotid( 9.00)
applying rotid: $1 :: (rot 0000000 $1) to 0777777
transform( 14.71): (rot 0000000 0777777) => (rot scount{7 8} tlatch{7 8})
operandmatch( 9.00)
decomposing by operand
transform( 2.55): 0000000 => scount{7 8}
(using previous result)
... success on transform( 2.55) with  2.00
transform( 12.16): 0777777 => tlatch{7 8}
applying fetch decomposition
search( 12.16): (<- tlatch{7 8} 0777777)
id.tl( 7.00)
feasible: id.tl = (<- tlatch{8 9999} abus{5 6})
transform( 11.16): 0777777 => abus{5 6}
applying fetch decomposition
search( 11.16): (<- abus{5 6} 0777777)
abus.fbus( 6.00)abus.dram( 8.00)
feasible: abus.fbus = (<- abus{5 12} fbus{2 3})
transform( 8.16): 0777777 => fbus{2 3}
applying fetch decomposition
search( 8.16): (<- fbus{2 3} 0777777)
fbus.ones( 3.00)
feasible: fbus.ones = (<- fbus{2 11} 0777777)
... squeezed out.
... cutoff reached.
... fail on search( 8.16)
... fail on transform( 8.16)
feasible: abus.dram = (<- abus{5 12} dram{4 5}[dadr{4 6} %wild])
transform( 8.16): 0777777 => dram{4 5}[dadr{4 6} %wild]
applying fetch decomposition
search( 8.16): (<- dram{4 5}[dadr{4 6} %wild] 0777777)
No takers!
... cutoff reached.
... fail on search( 8.16)
... fail on transform( 8.16)
... cutoff reached.
... fail on search( 11.16)
... fail on transform( 11.16)
... cutoff reached.
... fail on search( 12.16)
... fail on transform( 12.16)
... cutoff reached.
... fail on transform( 14.71)
... cutoff reached.
... fail on transform( 14.71)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777776 0777777)
transform( 14.71): (and 0777777 0777776) => (and %mask (rot scount{7 8} tlatch{7 8}))
andcommut( 9.00)con-unfold( 10.23)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 0777776)
transform( 14.71): (and 0777776 0777777) => (and %mask (rot scount{7 8} tlatch{7 8}))
... found previous failure
... fail on transform( 14.71)
applying con-unfold to (and 0777777 0777776)
transform( 14.71): (and 0777777 (rot 0000017 0077777)) => (and %mask (rot scount{7 8} tlatch{7 8}
andcommut( 10.23)operandmatch( 12.70)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 (rot 0000017 0077777))
transform( 14.71): (and (rot 0000017 0077777) 0777777) => (and %mask (rot scount{7 8} tlatch{7
andcommut( 10.23)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and (rot 0000017 0077777) 0777777)
transform( 14.71): (and 0777777 (rot 0000017 0077777)) => (and %mask (rot scount{7 8} tlatch{
... found previous failure
```

```
|... fail on transform( 14.71)
... cutoff reached.
... fail on transform( 14.71)
decomposing by operand
transform( 0.00): 0777777 => %mask
(using previous result)
... success on transform( 0.00) with  0.00
transform( 14.71): (rot 0000017 0077777) => (rot scount{7 8} tlatch{7 8})
No takers!
... cutoff reached.
... fail on transform( 14.71)
... cutoff reached.
... fail on transform( 14.71)
... cutoff reached.
... fail on transform( 14.71)
applying con-unfold to (and 0777776 0777777)
transform( 14.71): (and (rot 0000017 0077777) 0777777) => (and %mask (rot scount{7 8} tlatch{7 8}))
... found previous failure
... fail on transform( 14.71)
... cutoff reached.
... fail on transform( 14.71)
... cutoff reached.
... fail on search( 16.71)
... fail on transform( 16.71)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and (and 0777776 0777777) 0777777)
transform( 18.97): (and 0777777 (and 0777776 0777777)) => (and areg breg)
... found previous failure
... fail on transform( 18.97)
... cutoff reached.
... fail on transform( 18.97)
... cutoff reached.
... fail on transform( 18.97)
... cutoff reached.
... fail on transform( 18.97)
applying andid: $1 :: (and 0777777 $1) to (and 0777777 0777776)
transform( 18.97): (and 0777777 (and 0777777 0777776)) => (and areg breg)
andassoc2( 8.00)andcommut( 13.00)andid( 17.00)
applying andassoc2: (and $1 (and $2 $3)) :: (and (eval (and $1 $2)) $3) to (and 0777777 (and 0777777 0777776))
transform( 18.97): (and 0777777 0777776) => (and areg breg)
... found previous failure
... fail on transform( 18.97)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 (and 0777777 0777776))
transform( 18.97): (and (and 0777777 0777776) 0777777) => (and areg breg)
andassoc( 9.00)andcommut( 13.00)operandmatch( 13.00)
applying andassoc: (and (and $1 $2) $3) :: (and $1 (eval (and $2 $3))) to (and (and 0777777 0777776) 0777777
transform( 18.97): (and 0777777 0777776) => (and areg breg)
... found previous failure
... fail on transform( 18.97)
decomposing by operand
transform( 2.28): 0777777 => breg
(using previous result)
... success on transform( 2.28) with  2.00
transform( 16.71): (and 0777777 0777776) => areg
applying fetch decomposition
search( 16.71): (<- areg (and 0777777 0777776))
areg.mask( 16.17)
feasible: areg.mask = (<- areg(8 15) (and %mask (rot scount{7 8} tlatch{7 8})))
transform( 16.71): (and 0777777 0777776) => (and %mask (rot scount{7 8} tlatch{7 8}))
andcommut( 9.00)con-unfold( 10.23)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 0777776)
transform( 16.71): (and 0777776 0777777) => (and %mask (rot scount{7 8} tlatch{7 8}))
andcommut( 9.00)con-unfold( 10.23)operandmatch( 9.00)
decomposing by operand
transform( 0.00): 0777776 => %mask
(using previous result)
... success on transform( 0.00) with  0.00
transform( 16.71): 0777777 => (rot scount{7 8} tlatch{7 8})
rotid( 9.00)
applying rotid: $1 :: (rot 0000000 $1) to 0777777
transform( 16.71): (rot 0000000 0777777) => (rot scount{7 8} tlatch{7 8})
operandmatch( 9.00)
decomposing by operand
transform( 2.68): 0000000 => scount{7 8}
(using previous result)
... success on transform( 2.68) with  2.00
transform( 13.06): 0777777 => tlatch{7 8}
... found previous failure
... fail on transform( 13.06)
... cutoff reached.
... fail on transform( 16.71)
... cutoff reached.
... fail on transform( 16.71)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777776 0777777)
transform( 16.71): (and 0777777 0777776) => (and %mask (rot scount{7 8} tlatch{7 8}))
... found previous failure
... fail on transform( 16.71)
applying con-unfold to (and 0777776 0777777)
transform( 16.71): (and (rot 0000017 0077777) 0777777) => (and %mask (rot scount{7 8} tlatch{7 8}))
andcommut( 10.23)andid( 14.27)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and (rot 0000017 0077777) 0777777)
transform( 16.71): (and 0777777 (rot 0000017 0077777)) => (and %mask (rot scount{7 8} tlatch{7 8}
andcommut( 10.23)operandmatch( 12.70)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 (rot 0000017 0077777))
transform( 16.71): (and (rot 0000017 0077777) 0777777) => (and %mask (rot scount{7 8} tlatch{7
... found previous failure
... fail on transform( 16.71)
decomposing by operand
transform( 0.00): 0777777 => %mask
(using previous result)
... success on transform( 0.00) with  0.00
transform( 16.71): (rot 0000017 0077777) => (rot scount{7 8} tlatch{7 8})
No takers!
... cutoff reached.
... fail on transform( 16.71)
```

```
... cutoff reached.
... fail on transform( 15.71)
applying andid: $1 :: (and 0777777 $1) to (and (rot 0000017 0077777) 0777777)
transform( 15.71): (and 0777777 (and (rot 0000017 0077777) 0777777)) => (and %mask (rot scount{7
andassoc2( 9.00)
applying andassoc2: (and $1 (and $2 $3)) :: (and (eval (and $1 $2)) $3) to (and 0777777 (and (ro
transform( 15.71): (and 0777776 0777777) => (and %mask (rot scount{7 8} tlatch{7 8}))
... found previous failure
... fail on transform( 15.71)
... cutoff reached.
... fail on transform( 15.71)
... cutoff reached.
... fail on transform( 15.71)
... cutoff reached.
... fail on transform( 15.71)
applying con-unfold to (and 0777777 0777776)
transform( 15.71): (and 0777777 (rot 0000017 0077777)) => (and %mask (rot scount{7 8} tlatch{7 8}))
... found previous failure
... fail on transform( 15.71)
... cutoff reached.
... fail on transform( 15.71)
... cutoff reached.
... fail on search( 16.71)
... fail on transform( 16.71)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and (and 0777777 0777776) 0777777)
transform( 18.97): (and 0777777 (and 0777777 0777776)) => (and areg breg)
... found previous failure
... fail on transform( 18.97)
... cutoff reached.
... fail on transform( 18.97)
applying andid: $1 :: (and 0777777 $1) to (and 0777777 (and 0777777 0777776))
transform( 18.97): (and 0777777 (and 0777777 (and 0777777 0777776))) => (and areg breg)
andassoc2( 13.00)andcommut( 17.00)
applying andassoc2: (and $1 (and $2 $3)) :: (and (eval (and $1 $2)) $3) to (and 0777777 (and 0777777 (and 07
transform( 18.97): (and 0777777 (and 0777777 0777776)) => (and areg breg)
... found previous failure
... fail on transform( 18.97)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 (and 0777777 (and 0777777 0777776)))
transform( 18.97): (and (and 0777777 (and 0777777 0777776)) 0777777) => (and areg breg)
andassoc( 8.00)andcommut( 17.00)operandmatch( 17.00)
applying andassoc: (and (and $1 $2) $3) :: (and $1 (eval (and $2 $3))) to (and (and 0777777 (and 0777777 0
transform( 18.97): (and 0777777 0777776) => (and areg breg)
... found previous failure
... fail on transform( 18.97)
decomposing by operand
transform( 2.05): 0777777 => breg
(using previous result)
... success on transform( 2.05) with 2.00
transform( 16.92): (and 0777777 (and 0777777 0777776)) => areg
applying fetch decomposition
search( 16.92): (<- areg (and 0777777 (and 0777777 0777776)))
areg.mask( 12.06)
feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
transform( 14.92): (and 0777777 (and 0777777 0777776)) => (and %mask (rot scount{7 8} tlatch{7 8}))
andcommut( 10.59)andid( 14.19)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 (and 0777777 0777776))
transform( 14.92): (and (and 0777777 0777776) 0777777) => (and %mask (rot scount{7 8} tlatch{7 8}))
andcommut( 10.59)andid( 14.19)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and (and 0777777 0777776) 0777777)
transform( 14.92): (and 0777777 (and 0777777 0777776)) => (and %mask (rot scount{7 8} tlatch{7 8}
... found previous failure
... fail on transform( 14.92)
applying andid: $1 :: (and 0777777 $1) to (and (and 0777777 0777776) 0777777)
transform( 14.92): (and 0777777 (and (and 0777777 0777776) 0777777)) => (and %mask (rot scount{7
andcommut( 14.19)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 (and (and 0777777 0777776) 077777
transform( 14.92): (and (and (and 0777777 0777776) 0777777) 0777777) => (and %mask (rot scount{
andassoc( 10.59)
applying andassoc: (and (and $1 $2) $3) :: (and $1 (eval (and $2 $3))) to (and (and (and 07777
transform( 14.92): (and (and 0777777 0777776) 0777777) => (and %mask (rot scount{7 8} tlatch{
... found previous failure
... fail on transform( 14.92)
... cutoff reached.
... fail on transform( 14.92)
... cutoff reached.
... fail on transform( 14.92)
... cutoff reached.
... fail on transform( 14.92)
applying andid: $1 :: (and 0777777 $1) to (and 0777777 (and 0777777 0777776))
transform( 14.92): (and 0777777 (and 0777777 (and 0777777 0777776))) => (and %mask (rot scount{7 8}
andassoc2( 10.59)andcommut( 14.19)
applying andassoc2: (and $1 (and $2 $3)) :: (and (eval (and $1 $2)) $3) to (and 0777777 (and 07777
transform( 14.92): (and 0777777 (and 0777777 0777776)) => (and %mask (rot scount{7 8} tlatch{7 8}
... found previous failure
... fail on transform( 14.92)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 (and 0777777 (and 0777777 0777776))
transform( 14.92): (and (and 0777777 (and 0777777 0777776)) 0777777) => (and %mask (rot scount{7
andcommut( 14.19)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and (and 0777777 (and 0777777 0777776)) 07777
transform( 14.92): (and 0777777 (and 0777777 (and 0777777 0777776))) => (and %mask (rot scount{
... found previous failure
... fail on transform( 14.92)
... cutoff reached.
... fail on transform( 14.92)
... cutoff reached.
... fail on transform( 14.92)
... cutoff reached.
... fail on transform( 14.92)
... cutoff reached.
... fail on search( 16.92)
... fail on transform( 16.92)
applying andcommut: (and $1 $2) :: (and $2 $1) to (and (and 0777777 (and 0777777 0777776)) 0777777)
transform( 18.97): (and 0777777 (and 0777777 (and 0777777 0777776))) => (and areg breg)
... found previous failure
```

```
|  |  |  ... fail on transform( 18.97)
|  |  | ... cutoff reached.
|  |  | ... fail on transform( 18.97)
|  |  ... cutoff reached.
|  |  ... fail on transform( 18.97)
|  ... cutoff reached.
|  ... fail on transform( 18.97)
| decomposing by operand
|  transform( 6.82): 0777777 => areg
|   applying fetch decomposition
|    search( 6.82): (<- areg 0777777)
|     areg.mask( 6.00)
|     feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
|     |transform( 4.82): 0777777 => (and %mask (rot scount{7 8} tlatch{7 8}))
|     | No takers!
|     | ... cutoff reached.
|     | ... fail on transform( 4.82)
|     ... cutoff reached.
|     ... fail on search( 6.82)
|  ... fail on transform( 6.82)
|... cutoff reached.
|... fail on transform( 18.97)
... cutoff reached.
... fail on transform( 18.97)
feasible: fbus.or = (<- fbus{2 11} (or areg breg))
 transform( 18.97): 0777776 => (or areg breg)
  orid( 16.00)
  applying orid: $1 :: (or 0000000 $1) to 0777776
   transform( 18.97): (or 0000000 0777776) => (or areg breg)
   |orcommut( 12.00)operandmatch( 16.00)
   |applying orcommut: (or $1 $2) :: (or $2 $1) to (or 0000000 0777776)
    transform( 18.97): (or 0777776 0000000) => (or areg breg)
    orcommut( 16.00)operandmatch( 12.00)
    decomposing by operand
     transform( 9.49): 0777776 => areg
      ... found previous failure
      ... fail on transform( 9.49)
    applying orcommut: (or $1 $2) :: (or $2 $1) to (or 0777776 0000000)
     transform( 18.97): (or 0000000 0777776) => (or areg breg)
      ... found previous failure
      ... fail on transform( 18.97)
    ... cutoff reached.
    ... fail on transform( 18.97)
   decomposing by operand
    transform( 6.82): 0000000 => areg
     applying fetch decomposition
      search( 6.82): (<- areg 0000000)
       areg.mask( 6.00)
       feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
       |transform( 4.82): 0000000 => (and %mask (rot scount{7 8} tlatch{7 8}))
       | zeroand( 0.44)
       | applying zeroand: 0000000 :: (and 0000000 ???) to 0000000
       |  transform( 4.82): (and 0000000 ???) => (and %mask (rot scount{7 8} tlatch{7 8}))
       |   operandmatch( 0.00)
       |   decomposing by operand
       |    transform( 0.00): 0000000 => %mask
       |    |attempting constant match
       |    | it's a match!!
       |    |... success on transform( 0.00) with  0.00
       |    ... success on transform( 4.82) with  0.00
       | ... success on transform( 4.82) with  0.00
       ... success on search( 6.82) with  2.00
    ... success on transform( 6.82) with  2.00
   transform( 12.15): 0777776 => breg
    applying fetch decomposition
     search( 12.15): (<- breg 0777776)
      breg.con( 10.00)
      feasible: breg.con = (<- breg{4 13} (@2 0000010 conhi{3 4} conlo{3 4}))
      |transform( 10.15): 0777776 => (@2 0000010 conhi{3 4} conlo{3 4})
      | con-unfold( 8.00)
      | applying con-unfold to 0777776
      |  transform( 10.15): (@2 0000010 0000377 0000376) => (@2 0000010 conhi{3 4} conlo{3 4})
      |   operandmatch( 8.00)
      |   decomposing by operand
      |    transform( 5.07): 0000377 => conhi{3 4}
      |    |applying fetch decomposition
      |    | search( 5.07): (<- conhi{3 4} 0000377)
      |    |  ld.conhi( 4.00)
      |    |  feasible: ld.conhi = (<- conhi{0 9999} %wild)
      |    |   transform( 0.00): 0000377 => %wild
      |    |    attempting constant match
      |    |    |it's a match!!
      |    |    ... success on transform( 0.00) with  0.00
      |    |   ... success on search( 5.07) with  4.00
      |    |... success on transform( 5.07) with  4.00
      |    transform( 5.07): 0000376 => conlo{3 4}
      |    |applying fetch decomposition
      |    | search( 5.07): (<- conlo{3 4} 0000376)
      |    |  ld.conlo( 4.00)
      |    |  feasible: ld.conlo = (<- conlo{0 9999} %wild)
      |    |   ... squeezed out.
      |    |   ... cutoff reached.
      |    |   ... fail on search( 5.07)
      |    |... fail on transform( 5.07)
      |    ... cutoff reached.
      |    ... fail on transform( 10.15)
      | ... cutoff reached.
      | ... fail on transform( 10.15)
      ... cutoff reached.
      ... fail on search( 12.15)
   ... fail on transform( 12.15)
|... cutoff reached.
|... fail on transform( 18.97)
... cutoff reached.
```

```
... fail on transform( 18.97)
feasible: fbus.xor = (<- fbus{2 11} (xor areg breg))
 transform( 18.97): 0777776 => (xor areg breg)
  xorid( 12.00)
  applying xorid: $1 :: (xor 0000000 $1) to 0777776
   transform( 18.97): (xor 0000000 0777776) => (xor areg breg)
    xorcommut( 12.00)operandmatch( 12.00)
    decomposing by operand
     transform( 2.36): 0000000 => areg
       (using previous result)
      ... success on transform( 2.36) with  2.00
     transform( 16.61): 0777776 => breg
      applying fetch decomposition
       search( 16.61): (<- breg 0777776)
        breg.fbl( 16.00)breg.con( 10.00)
        feasible: breg.con = (<- breg{4 13} (@2 0000010 conhi{3 4} conlo{3 4}))
        transform( 14.61): 0777776 => (@2 0000010 conhi{3 4} conlo{3 4})
         con-unfold( 8.00)
         applying con-unfold to 0777776
          transform( 14.61): (@2 0000010 0000377 0000376) => (@2 0000010 conhi{3 4} conlo{3 4})
           operandmatch( 8.00)
           decomposing by operand
            transform( 7.30): 0000377 => conhi{3 4}
             (using previous result)
            ... success on transform( 7.30) with  4.00
            transform( 7.30): 0000376 => conlo{3 4}
            applying fetch decomposition
             search( 7.30): (<- conlo{3 4} 0000376)
              ld.conlo( 4.00)
              feasible: ld.conlo = (<- conlo{0 9999} %wild)
               ... squeezed out.
              ... cutoff reached.
              ... fail on search( 7.30)
             ... fail on transform( 7.30)
           ... cutoff reached.
           ... fail on transform( 14.61)
         ... cutoff reached.
         ... fail on transform( 14.61)
        feasible: breg.fbl = (<- breg{4 13} fblatch{3 4})
        transform( 14.61): 0777776 => fblatch{3 4}
         applying fetch decomposition
          search( 14.61): (<- fblatch{3 4} 0777776)
           ld.fbl( 14.00)
           feasible: ld.fbl = (<- fblatch{3 9999} fbus{2 3})
            transform( 13.61): 0777776 => fbus{2 3}
            applying fetch decomposition
             search( 13.61): (<- fbus{2 3} 0777776)
              ... found previous failure
              ... fail on search( 13.61)
             ... fail on transform( 13.61)
           ... cutoff reached.
           ... fail on search( 14.61)
         ... fail on transform( 14.61)
        ... cutoff reached.
        ... fail on search( 16.61)
     ... fail on transform( 16.61)
    applying xorcommut: (xor $1 $2) :: (xor $2 $1) to (xor 0000000 0777776)
     transform( 13.97): (xor 0777776 0000000) => (xor areg breg)
      xorcommut( 12.00)operandmatch( 12.00)
      decomposing by operand
       transform( 9.49): 0777776 => areg
        ... found previous failure
        ... fail on transform( 9.49)
      applying xorcommut: (xor $1 $2) :: (xor $2 $1) to (xor 0777776 0000000)
       transform( 18.97): (xor 0000000 0777776) => (xor areg breg)
        ... found previous failure
        ... fail on transform( 18.97)
      ... cutoff reached.
      ... fail on transform( 18.97)
    ... cutoff reached.
    ... fail on transform( 18.97)
  ... cutoff reached.
  ... fail on transform( 18.97)
feasible: fbus.add = (<- fbus{2 11} (+ (+ areg breg) carryin))
 transform( 18.97): 0777776 => (+ (+ areg breg) carryin)
  con-unfold( 12.00)plusid( 16.00)p3carylid( 16.00)
  applying con-unfold to 0777776
   transform( 18.97): (+ 0777777 0777777) => (+ (+ areg breg) carryin)
   plusid( 10.00)pluscommut( 12.00)con-unfold( 12.80)
   applying plusid: $1 :: (+ 0000000 $1) to (+ 0777777 0777777)
    transform( 18.97): (+ 0000000 (+ 0777777 0777777)) => (+ (+ areg breg) carryin)
     pluscommut( 10.00)plusassoc2( 12.00)p3carylid( 16.00)
     applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0000000 (+ 0777777 0777777))
      transform( 18.97): (+ (+ 0777777 0777777) 0000000) => (+ (+ areg breg) carryin)
       pluscommut( 10.00)plusassoc( 12.00)operandmatch( 10.00)
       decomposing by operand
       transform( 2.69): 0000000 => carryin
       applying fetch decomposition
        search( 2.69): (<- carryin 0000000)
         carry.0( 2.00)
         feasible: carry.0 = (<- carryin{0 9} 0000000)
         ... success on search( 2.69) with  2.00
       ... success on transform( 2.69) with  2.00
       transform( 16.28): (+ 0777777 0777777) => (+ areg breg)
       pluscommut( 8.00)p3carylid( 16.00)operandmatch( 8.00)
       decomposing by operand
        transform( 3.04): 0777777 => breg
          (using previous result)
         ... success on transform( 3.04) with  2.00
        transform( 13.25): 0777777 => areg
         applying fetch decomposition
          search( 13.25): (<- areg 0777777)
           areg.mask( 6.00)
           feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
```

```
| | | transform( 11.25): 0777777 => (and %mask (rot scount{7 8} tlatch{7 8}))
| | | No takers!
| | | ... cutoff reached.
| | | ... fail on transform( 11.25)
| | |... cutoff reached.
| | |... fail on search( 13.25)
| | ... fail on transform( 13.25)
| | applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0777777 0777777)
| | transform( 16.28): (+ 0777777 0777777) => (+ areg breg)
| | ... found previous failure
| | ... fail on transform( 16.28)
| | applying p3carylid: $1 :: (+ (eval (+ 0777777 $1)) 0000001) to (+ 0777777 0777777)
| | transform( 16.28): (+ 0777775 0000001) => (+ areg breg)
| | pluscommut( 16.00)p3carylid( 16.00)
| | applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0777775 0000001)
| | transform( 16.28): (+ 0000001 0777775) => (+ areg breg)
| | |pluscommut( 16.00)p3carylid( 16.00)operandmatch( 16.00)
| | |decomposing by operand
| | | transform( 6.08): 0000001 => areg
| | | applying fetch decomposition
| | | search( 6.08): (<- areg 0000001)
| | | areg.mask( 6.00)
| | | feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
| | | |transform( 4.08): 0000001 => (and %mask (rot scount{7 8} tlatch{7 8}))
| | | | No takers!
| | | | ... cutoff reached.
| | | | ... fail on transform( 4.08)
| | | ... cutoff reached.
| | | ... fail on search( 6.08)
| | | ... fail on transform( 6.08)
| | | applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0000001 0777775)
| | | transform( 16.28): (+ 0777775 0000001) => (+ areg breg)
| | | ... found previous failure
| | | ... fail on transform( 16.28)
| | | applying p3carylid: $1 :: (+ (eval (+ 0777777 $1)) 0000001) to (+ 0000001 0777775)
| | | transform( 16.28): (+ 0777775 0000001) => (+ areg breg)
| | | ... found previous failure
| | | ... fail on transform( 16.28)
| | |... cutoff reached.
| | |... fail on transform( 16.28)
| | ... applying p3carylid: $1 :: (+ (eval (+ 0777777 $1)) 0000001) to (+ 0777775 0000001)
| | transform( 16.28): (+ 0777775 0000001) => (+ areg breg)
| | |... found previous failure
| | |... fail on transform( 16.28)
| | ... cutoff reached.
| | ... fail on transform( 16.28)
| | ... cutoff reached.
| | ... fail on transform( 16.28)
| | applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ (+ 0777777 0777777) 0000000)
| | transform( 18.97): (+ 0000000 (+ 0777777 0777777)) => (+ (+ areg breg) carryin)
| | ... found previous failure
| | ... fail on transform( 18.97)
| | applying plusassoc: (+ (+ $1 $2) $3) :: (+ $1 (eval (+ $2 $3))) to (+ (+ 0777777 0777777) 0000000)
| | transform( 18.97): (+ 0777777 0777777) => (+ (+ areg breg) carryin)
| | ... found previous failure
| | ... fail on transform( 18.97)
| | ... cutoff reached.
| | ... fail on transform( 18.97)
| | applying plusassoc2: (+ $1 (+ $2 $3)) :: (+ (eval (+ $1 $2)) $3) to (+ 0000000 (+ 0777777 0777777))
| | transform( 18.97): (+ 0777777 0777777) => (+ (+ areg breg) carryin)
| | ... found previous failure
| | ... fail on transform( 18.97)
| | applying p3carylid: $1 :: (+ (eval (+ 0777777 $1)) 0000001) to (+ 0000000 (+ 0777777 0777777))
| | transform( 18.97): (+ 0777775 0000001) => (+ (+ areg breg) carryin)
| | con-unfold( 10.00)p3carylid( 16.00)operandmatch( 16.00)
| | applying con-unfold to (+ 0777775 0000001)
| | transform( 18.97): (+ (+ 0777776 0777777) 0000001) => (+ (+ areg breg) carryin)
| | plusassoc( 16.00)p3carylid( 16.00)operandmatch( 16.00)
| | decomposing by operand
| | transform( 2.69): 0000001 => carryin
| | applying fetch decomposition
| | search( 2.69): (<- carryin 0000001)
| | carry.1( 2.00)
| | feasible: carry.1 = (<- carryin{0 0} 0000001)
| | ... success on search( 2.69) with 2.00
| | ... success on transform( 2.69) with 2.00
| | transform( 16.28): (+ 0777776 0777777) => (+ areg breg)
| | p3carylid( 16.00)pluscommut( 16.13)
| | applying p3carylid: $1 :: (+ (eval (+ 0777777 $1)) 0000001) to (+ 0777776 0777777)
| | transform( 16.28): (+ 0777774 0000001) => (+ areg breg)
| | |pluscommut( 16.00)p3carylid( 16.00)operandmatch( 16.00)
| | |decomposing by operand
| | | transform( 6.08): 0777774 => areg
| | | applying fetch decomposition
| | | search( 6.08): (<- areg 0777774)
| | | areg.mask( 6.00)
| | | feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
| | | |transform( 4.08): 0777774 => (and %mask (rot scount{7 8} tlatch{7 8}))
| | | | No takers!
| | | | ... cutoff reached.
| | | | ... fail on transform( 4.08)
| | | ... cutoff reached.
| | | ... fail on search( 6.08)
| | | ... fail on transform( 6.08)
| | | applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0777774 0000001)
| | | transform( 16.28): (+ 0000001 0777774) => (+ areg breg)
| | | pluscommut( 16.00)operandmatch( 16.00)
| | | decomposing by operand
| | | transform( 6.08): 0000001 => areg
| | | ... found previous failure
| | | ... fail on transform( 6.08)
| | | applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0000001 0777774)
| | | transform( 16.28): (+ 0777774 0000001) => (+ areg breg)
| | | ... found previous failure
```

```
... fail on transform( 16.28)
... cutoff reached.
... fail on transform( 16.28)
applying p3carylid: $1 :: (+ (eval (+ 0777777 $1)) 0000001) to (+ 0777774 0000001)
transform( 16.28): (+ 0777774 0000001) => (+ areg breg)
... found previous failure
... fail on transform( 16.28)
... cutoff reached.
... fail on transform( 16.28)
applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0777776 0777777)
transform( 16.28): (+ 0777777 0777776) => (+ areg breg)
pluscommut( 8.00)con-unfold( 16.13)
applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0777777 0777776)
transform( 16.28): (+ 0777776 0777777) => (+ areg breg)
... found previous failure
... fail on transform( 16.28)
applying con-unfold to (+ 0777777 0777776)
transform( 16.28): (+ 0777777 (+ 0777777 0777777)) => (+ areg breg)
plusassoc2( 8.00)
applying plusassoc2: (+ $1 (+ $2 $3)) :: (+ (eval (+ $1 $2)) $3) to (+ 0777777 (+ 0777777 0777777))
transform( 16.28): (+ 0777776 0777777) => (+ areg breg)
... found previous failure
... fail on transform( 16.28)
... cutoff reached.
... fail on transform( 16.28)
... cutoff reached.
... fail on transform( 16.28)
... cutoff reached.
... fail on transform( 16.28)
applying plusassoc: (+ (+ $1 $2) $3) :: (+ $1 (eval (+ $2 $3))) to (+ (+ 0777776 0777777) 0000001)
transform( 18.97): (+ 0777776 0000000) => (+ (+ areg breg) carryin)
plusid( 14.00)pluscommut( 16.00)operandmatch( 16.00)
applying plusid: $1 :: (+ 0000000 $1) to (+ 0777776 0000000)
transform( 18.97): (+ 0000000 (+ 0777776 0000000)) => (+ (+ areg breg) carryin)
pluscommut( 14.00)
applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0000000 (+ 0777776 0000000))
transform( 18.97): (+ (+ 0777776 0000000) 0000000) => (+ (+ areg breg) carryin)
pluscommut( 14.00)operandmatch( 14.00)
decomposing by operand
transform( 2.18): 0000000 => carryin
(using previous result)
... success on transform( 2.18) with  2.00
transform( 16.79): (+ 0777776 0000000) => (+ areg breg)
pluscommut( 16.18)operandmatch( 12.00)
decomposing by operand
transform( 8.39): 0777776 => areg
... found previous failure
... fail on transform( 8.39)
applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0777776 0000000)
transform( 16.79): (+ 0000000 0777776) => (+ areg breg)
pluscommut( 12.00)
applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0000000 0777776)
transform( 16.79): (+ 0777776 0000000) => (+ areg breg)
... found previous failure
... fail on transform( 16.79)
... cutoff reached.
... fail on transform( 16.79)
... cutoff reached.
... fail on transform( 16.79)
applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ (+ 0777776 0000000) 0000000)
transform( 18.97): (+ 0000000 (+ 0777776 0000000)) => (+ (+ areg breg) carryin)
... found previous failure
... fail on transform( 18.97)
... cutoff reached.
... fail on transform( 18.97)
... cutoff reached.
... fail on transform( 18.97)
decomposing by operand
transform( 2.08): 0000000 => carryin
(using previous result)
... success on transform( 2.08) with  2.00
transform( 16.89): 0777776 => (+ areg breg)
No takers!
... cutoff reached.
... fail on transform( 16.89)
applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0777776 0000000)
transform( 18.97): (+ 0000000 0777776) => (+ (+ areg breg) carryin)
con-unfold( 10.00)
applying con-unfold to (+ 0000000 0777776)
transform( 18.97): (+ 0000000 (+ 0777777 0777777)) => (+ (+ areg breg) carryin)
... found previous failure
... fail on transform( 18.97)
... cutoff reached.
... fail on transform( 18.97)
... cutoff reached.
... fail on transform( 18.97)
applying p3carylid: $1 :: (+ (eval (+ 0777777 $1)) 0000001) to (+ (+ 0777776 0777777) 0000001)
transform( 18.97): (+ 0777775 0000001) => (+ (+ areg breg) carryin)
... found previous failure
... fail on transform( 18.97)
... cutoff reached.
... fail on transform( 18.97)
decomposing by operand
transform( 2.08): 0000001 => carryin
(using previous result)
... success on transform( 2.08) with  2.00
transform( 16.89): 0777775 => (+ areg breg)
plusid( 12.00)
applying plusid: $1 :: (+ 0000000 $1) to 0777775
transform( 16.89): (+ 0000000 0777775) => (+ areg breg)
operandmatch( 12.00)
decomposing by operand
transform( 2.25): 0000000 => areg
(using previous result)
```

```
|   |   |... success on transform( 2.25) with  2.00
|   |   transform( 14.63): 0777775 => breg
|   |   |applying fetch decomposition
|   |   | search( 14.63): (<- breg 0777775)
|   |   |  breg.con( 10.00)
|   |   |  feasible: breg.con = (<- breg{4 13} (@2 0000010 conhi{3 4} conlo{3 4}))
|   |   |   transform( 12.63): 0777775 => (@2 0000010 conhi{3 4} conlo{3 4})
|   |   |    con-unfold( 8.00)
|   |   |    applying con-unfold to 0777775
|   |   |    |transform( 12.63): (@2 0000010 0000377 0000375) => (@2 0000010 conhi{3 4} conlo{3 4})
|   |   |    | operandmatch( 8.00)
|   |   |    | decomposing by operand
|   |   |    |  transform( 6.32): 0000377 => conhi{3 4}
|   |   |    |    (using previous result)
|   |   |    |    ... success on transform( 6.32) with  4.00
|   |   |    |  transform( 6.32): 0000375 => conlo{3 4}
|   |   |    |   applying fetch decomposition
|   |   |    |    search( 6.32): (<- conlo{3 4} 0000375)
|   |   |    |    |ld.conlo( 4.00)
|   |   |    |    |feasible: ld.conlo = (<- conlo{0 9999} %wild)
|   |   |    |    | ... squeezed out.
|   |   |    |    |... cutoff reached.
|   |   |    |    |... fail on search( 6.32)
|   |   |    |   ... fail on transform( 6.32)
|   |   |    |  ... cutoff reached.
|   |   |    |  ... fail on transform( 12.63)
|   |   |    ... cutoff reached.
|   |   |    ... fail on transform( 12.63)
|   |   |   ... cutoff reached.
|   |   |   ... fail on search( 14.63)
|   |   |... fail on transform( 14.63)
|   |  ... cutoff reached.
|   |  ... fail on transform( 16.89)
|   | ... cutoff reached.
|   | ... fail on transform( 16.89)
|   applying p3carry1id: $1 :: (+ (eval (+ 0777777 $1)) 0000001) t  (+ 0777775 0000001)
|   |transform( 18.97): (+ 0777775 0000001) => (+ (+ areg breg) carryin)
|   | ... found previous failure
|   | ... fail on transform( 18.97)
|   ... cutoff reached.
|   ... fail on transform( 18.97)
|  ... cutoff reached.
|  ... fail on transform( 18.97)
| applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0777777 0777777)
|  transform( 18.97): (+ 0777777 0777777) => (+ (+ areg breg) carryin)
|   ... found previous failure
|   ... fail on transform( 18.97)
| applying con-unfold to (+ 0777777 0777777)
|  transform( 18.97): (+ (+ 0000000 0777777) 0777777) => (+ (+ areg breg) carryin)
|   pluscommut( 12.80)plusid( 16.40)con-unfold( 16.40)
|   applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ (+ 0000000 0777777) 0777777)
|    transform( 18.97): (+ 0777777 (+ 0000000 0777777)) => (+ (+ areg breg) carryin)
|     plusassoc2( 12.00)pluscommut( 12.80)
|     applying plusassoc2: (+ $1 (+ $2 $3)) :: (+ (eval (+ $1 $2)) $3) to (+ 0777777 (+ 0000000 0777777))
|     |transform( 18.97): (+ 0777777 0777777) => (+ (+ areg breg) carryin)
|     | ... found previous failure
|     | ... fail on transform( 18.97)
|     applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0777777 (+ 0000000 0777777))
|     |transform( 18.97): (+ (+ 0000000 0777777) 0777777) => (+ (+ areg breg) carryin)
|     | ... found previous failure
|     | ... fail on transform( 18.97)
|     ... cutoff reached.
|     ... fail on transform( 18.97)
|   applying plusid: $1 :: (+ 0000000 $1) to (+ (+ 0000000 0777777) 0777777)
|    transform( 18.97): (+ 0000000 (+ (+ 0000000 0777777) 0777777)) => (+ (+ areg breg) carryin)
|     plusassoc2( 12.00)pluscommut( 16.40)
|     applying plusassoc2: (+ $1 (+ $2 $3)) :: (+ (eval (+ $1 $2)) $3) to (+ 0000000 (+ (+ 0000000 0777777) 077777
|     |transform( 18.97): (+ 0777777 0777777) => (+ (+ areg breg) carryin)
|     | ... found previous failure
|     | ... fail on transform( 18.97)
|     applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0000000 (+ (+ 0000000 0777777) 0777777))
|     |transform( 18.97): (+ (+ (+ 0000000 0777777) 0777777) 0000000) => (+ (+ areg breg) carryin)
|     | plusassoc( 12.80)pluscommut( 16.40)
|     | applying plusassoc: (+ (+ $1 $2) $3) :: (+ $1 (eval (+ $2 $3))) to (+ (+ (+ 0000000 0777777) 0777777) 0000
|     |  transform( 18.97): (+ (+ 0000000 0777777) 0777777) => (+ (+ areg breg) carryin)
|     |  | ... found previous failure
|     |  | ... fail on transform( 18.97)
|     | applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ (+ (+ 0000000 0777777) 0777777) 0000000)
|     |  transform( 18.97): (+ 0000000 (+ (+ 0000000 0777777) 0777777)) => (+ (+ areg breg) carryin)
|     |  | ... found previous failure
|     |  | ... fail on transform( 18.97)
|     | ... cutoff reached.
|     | ... fail on transform( 18.97)
|     ... cutoff reached.
|     ... fail on transform( 18.97)
|   applying con-unfold to (+ (+ 0000000 0777777) 0777777)
|    transform( 18.97): (+ (+ 0000000 0777777) (+ 0000000 0777777)) => (+ (+ areg breg) carryin)
|     plusassoc2( 12.00)pluscommut( 16.40)
|     applying plusassoc2: (+ $1 (+ $2 $3)) :: (+ (eval (+ $1 $2)) $3) to (+ (+ 0000000 0777777) (+ 0000000 077777
|     |transform( 18.97): (+ 0777777 0777777) => (+ (+ areg breg) carryin)
|     | ... found previous failure
|     | ... fail on transform( 18.97)
|     applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ (+ 0000000 0777777) (+ 0000000 0777777))
|     |transform( 18.97): (+ (+ 0000000 0777777) (+ 0000000 0777777)) => (+ (+ areg breg) carryin)
|     | ... found previous failure
|     | ... fail on transform( 18.97)
|     ... cutoff reached.
|     ... fail on transform( 18.97)
|   ... cutoff reached.
|   ... fail on transform( 18.97)
|  ... cutoff reached.
|  ... fail on transform( 18.97)
| applying plusid: $1 :: (+ 0000000 $1) to 0777776
|  transform( 18.97): (+ 0000000 0777776) => (+ (+ areg breg) carryin)
```

```
    |... found previous failure
    |... fail on transform( 18.97)
  applying p3carylid: $1 :: (+ (eval (+ 0777777 $1)) 0000001) to 0777776
   transform( 18.97): (+ 0777775 0000001) => (+ (+ areg breg) carryin)
    |... found previous failure
    |... fail on transform( 18.97)
  ... cutoff reached.
  ... fail on transform( 18.97)
feasible: fbus.bma = (<- fbus{2 11} (+ (+ (not areg) breg) carryin))
 transform( 18.97): 0777776 => (+ (+ (not areg) breg) carryin)
  con-unfold( 14.00)plusid( 16.00)con-unfold( 18.02)
  applying con-unfold to 0777776
   transform( 18.97): (not 0000001) => (+ (+ (not areg) breg) carryin)
   |plusid( 12.00)con-unfold( 16.40)
   |applying plusid: $1 :: (+ 0000000 $1) to (not 0000001)
    transform( 18.97): (+ 0000000 (not 0000001)) => (+ (+ (not areg) breg) carryin)
     pluscommut( 12.00)plusid( 12.20)con-unfold( 12.80)
     applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0000000 (not 0000001))
      transform( 18.97): (+ (not 0000001) 0000000) => (+ (+ (not areg) breg) carryin)
       pluscommut( 12.00)plusid( 12.20)operandmatch( 12.00)
       decomposing by operand
       |transform( 2.36): 0000000 => carryin
         (using previous result)
        ... success on transform( 2.36) with  2.00
       |transform( 16.61): (not 0000001) => (+ (not areg) breg)
        plusid( 12.00)con-unfold( 16.16)
        applying plusid: $1 :: (+ 0000000 $1) to (not 0000001)
         transform( 16.61): (+ 0000000 (not 0000001)) => (+ (not areg) breg)
          pluscommut( 12.00)
          applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0000000 (not 0000001))
           transform( 16.61): (+ (not 0000001) 0000000) => (+ (not areg) breg)
            |pluscommut( 12.00)operandmatch( 12.00)
            |decomposing by operand
             transform( 8.30): (not 0000001) => (not areg)
              operandmatch( 6.00)
              decomposing by operand
               transform( 8.30): 0000001 => areg
                applying fetch decomposition
                |search( 8.30): (<- areg 0000001)
                | areg.mask( 6.00)
                | feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
                |  transform( 6.30): 0000001 => (and %mask (rot scount{7 8} tlatch{7 8}))
                |   No takers!
                |    ... cutoff reached.
                |    ... fail on transform( 6.30)
                | ... cutoff reached.
                | ... fail on search( 8.30)
                ... fail on transform( 8.30)
              ... cutoff reached.
              ... fail on transform( 8.30)
            |applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ (not 0000001) 0000000)
            | transform( 16.61): (+ 0000000 (not 0000001)) => (+ (not areg) breg)
            |  ... found previous failure
            |  ... fail on transform( 16.61)
            |... cutoff reached.
            |... fail on transform( 16.61)
           ... cutoff reached.
           ... fail on transform( 16.61)
          applying con-unfold to (not 0000001)
           transform( 16.61): (not (+ 0000000 0000001)) => (+ (not areg) breg)
            No takers!
            ... cutoff reached.
            ... fail on transform( 16.61)
         ... cutoff reached.
         ... fail on transform( 16.61)
        applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ (not 0000001) 0000000)
        |transform( 18.97): (+ 0000000 (not 0000001)) => (+ (+ (not areg) breg) carryin)
         ... found previous failure
         ... fail on transform( 18.97)
        applying plusid: $1 :: (+ 0000000 $1) to (+ (not 0000001) 0000000)
        |transform( 18.97): (+ 0000000 (+ (not 0000001) 0000000)) => (+ (+ (not areg) breg) carryin)
         pluscommut( 12.20)con-unfold( 14.60)
         applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ 0000000 (+ (not 0000001) 0000000))
          transform( 18.97): (+ (+ (not 0000001) 0000000) 0000000) => (+ (+ (not areg) breg) carryin)
           plusassoc( 12.00)pluscommut( 12.20)operandmatch( 14.00)
           applying plusassoc: (+ (+ $1 $2) $3) :: (+ $1 (eval (+ $2 $3))) to (+ (+ (not 0000001) 0000000) 0000000)
            transform( 18.97): (+ (not 0000001) 0000000) => (+ (+ (not areg) breg) carryin)
             |... found previous failure
             |... fail on transform( 18.97)
            applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ (+ (not 0000001) 0000000) 0000000)
             transform( 18.97): (+ 0000000 (+ (not 0000001) 0000000)) => (+ (+ (not areg) breg) carryin)
              |... found previous failure
              |... fail on transform( 18.97)
            decomposing by operand
             transform( 2.18): 0000000 => carryin
              (using previous result)
              |... success on transform( 2.18) with  2.00
             transform( 16.79): (+ (not 0000001) 0000000) => (+ (not areg) breg)
              |... found previous failure
              |... fail on transform( 16.79)
            ... cutoff reached.
            ... fail on transform( 18.97)
           applying con-unfold to (+ 0000000 (+ (not 0000001) 0000000))
            transform( 18.97): (+ (not 0777777) (+ (not 0000001) 0000000)) => (+ (+ (not areg) breg) carryin)
             pluscommut( 14.60)plusassoc2( 16.00)
             applying pluscommut: (+ $1 $2) :: (+ $2 $1) to (+ (not 0777777) (+ (not 0000001) 0000000))
              transform( 18.97): (+ (+ (not 0000001) 0000000) (not 0777777)) => (+ (+ (not areg) breg) carryin)
              |plusassoc( 12.00)
              |applying plusassoc: (+ (+ $1 $2) $3) :: (+ $1 (eval (+ $2 $3))) to (+ (+ (not 0000001) 0000000) (not 0
              | transform( 18.97): (+ (not 0000001) 0000000) => (+ (+ (not areg) breg) carryin)
              |  ... found previous failure
              |  ... fail on transform( 18.97)
              | ... cutoff reached.
              | ... fail on transform( 18.97)
```

```
|  |  applying plusassoc2: (+ $1 (+ $2 $3)) :: (+ (eval (+ $1 $2)) $3) to (+ (not 0777777) (+ (not 0000001) 00
|  |  transform( 18.97): (+ 0777776 0000000) => (+ (+ (not areg) breg) carryin)
|  |  |con-unfold( 12.00)operandmatch( 16.00)
|  |  |applying con-unfold to (+ 0777776 0000000)
|  |  | transform( 18.97): (+ (+ 0777777 0777777) 0000000) => (+ (+ (not areg) breg) carryin)
|  |  |  con-unfold( 12.80)operandmatch( 12.00)
|  |  |  decomposing by operand
|  |  |   transform( 2.36): 0000000 => carryin
|  |  |    |(using previous result)
|  |  |    ... success on transform( 2.36) with  2.00
|  |  |   transform( 16.61): (+ 0777777 0777777) => (+ (not areg) breg)
|  |  |    con-unfold( 4.00)operandmatch(  4.00)
|  |  |    applying con-unfold to (+ 0777777 0777777)
|  |  |    |transform( 16.61): (+ (not 0000000) 0777777) => (+ (not areg) breg)
|  |  |    | pluscommut( 4.00)operandmatch( 4.00)
|  |  |    | decomposing by operand
|  |  |    |  transform( 8.30): (not 0000000) => (not areg)
|  |  |    |   operandmatch( 2.00)
|  |  |    |   decomposing by operand
|  |  |    |    transform( 8.30): 0000000 => areg
|  |  |    |    | (using previous result)
|  |  |    |    |... success on transform( 8.30) with  2.00
|  |  |    |   ... success on transform( 8.30) with  2.00
|  |  |    |  transform( 8.30): 0777777 => breg
|  |  |    |   (using previous result)
|  |  |    |   ... success on transform( 8.30) with  2.00
|  |  |    | ... success on transform( 16.61) with  4.00
|  |  |    ... success on transform( 16.61) with  4.00
|  |  |   ... success on transform( 18.97) with  6.00
|  |  |  ... success on transform( 18.97) with  6.00
|  |  ... success on transform( 18.97) with  6.00
|  | ... success on transform( 18.97) with  6.00
|  ... success on transform( 18.97) with  6.00
| ... success on transform( 18.97) with  6.00
... success on transform( 18.97) with  6.00
... success on search( 21.97) with  9.00
253 nodes examined.
Maximum search depth: 17
Maximum axiom depth: 9
Approximate execution time: 116.68 seconds

Compacting:
areg.mask 0000000 breg.ones  (0)
fbus.bma  carry.0  (1)
```

In the final trace, the *And/Or* and *iteration* strategies are used together.

```
search( 45.60): (; (<- dram[dadr 0000000] lincwd) (<- fbus 0000007))
 decomposition( 38.00)
 search( 20.40): (<- dram[dadr 0000000] lincwd)
  ld.dr.aset( 17.00)ld.dr.aclr( 17.00)
  feasible  ld.dr.aset = (<- dram{3 9999}[dadr{2 3} %wild] (or dmask{1 2} abus))
   transform( 0.00)*: dram{3 9999}[dadr{2 3} %wild] => dram[dadr 0000000]
    transform( 0.00): 0000000 => %wild
     |attempting constant match
     | it's a match!!
     |... success on transform( 0.00) with  0.00
    ... success on transform( 0.00) with  0.00
   transform( 18.40): lincwd => (or dmask{1 2} abus)
    orid( 7.00)
    applying orid: $1 :: (or 0000000 $1) to lincwd
    |transform( 18.40): (or 0000000 lincwd) => (or dmask{1 2} abus)
    | operandmatch( 7.00)
    | decomposing by operand
    |  transform( 0.00): 0000000 => dmask{1 2}
    |   applying fetch decomposition
    |    search( 0.00): (<- dmask{1 2} 0000000)
    |    |ld.dmask( 0.00)
    |    |feasible: ld.dmask = (<- dmask{0 7} %bitset)
    |    | transform( 0.00): 0000000 => %bitset
    |    |  attempting constant match
    |    |   it's a match!!
    |    |   ... success on transform( 0.00) with  0.00
    |    |... success on search( 0.00) with  0.00
    |   ... success on transform( 0.00) with  0.00
    |  transform( 18.40): lincwd => abus
    |   applying fetch decomposition
    |    search( 18.40): (<- abus lincwd)
    |    |abus.linc( 7.00)
    |    |feasible: abus.linc = (<- abus{5 12} lincwd{4 5})
    |    |... success on search( 18.40) with  7.00
    |   ... success on transform( 18.40) with  7.00
    | ... success on transform( 18.40) with  7.00
    ... success on transform( 18.40) with  7.00
  feasible  ld.dr.aclr = (<- dram{3 9999}[dadr{2 3} %wild] (and (not dmask{1 2}) abus))
   transform( 0.00)*: dram{3 9999}[dadr{2 3} %wild] => dram[dadr 0000000]
    |(using previous result)
    ... success on transform( 0.00) with  0.00
   transform( 16.14): lincwd => (and (not dmask{1 2}) abus)
    No takers!
    ... cutoff reached.
    ... fail on transform( 16.14)
  ... success on search( 20.40) with  9.00
 search( 25.20): (<- fbus 0000007)
  fbus.and( 21.00)fbus.or( 21.00)fbus.xor( 21.00)
  feasible: fbus.and = (<- fbus{2 11} (and areg breg))
   transform( 22.20): 0000007 => (and areg breg)
    andid( 12.00)
    applying andid: $1 :: (and 0777777 $1) to 0000007
```

```
|transform( 22.20): (and 0777777 0000007) => (and areg breg)
 andcommut( 12.00)andid( 20.00)
 applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 0000007)
  transform( 22.20): (and 0000007 0777777) => (and areg breg)
   andcommut( 12.00)andid( 20.00)operandmatch( 12.00)
   decomposing by operand
    transform( 2.53): 0777777 => breg
    |applying fetch decomposition
     search( 2.53): (<- breg 0777777)
      breg.ones( 2.00)
      feasible: breg.ones = (<- breg{4 13} 0777777)
      ... success on search( 2.53) with  2.00
    |... success on transform( 2.53) with  2.00
    transform( 19.67): 0000007 => areg
    |applying fetch decomposition
     search( 19.67): (<- areg 0000007)
      areg.mask( 10.00)
      feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
       transform( 17.67): 0000007 => (and %mask (rot scount{7 8} tlatch{7 8}))
        andid( 11.00)
        applying andid: $1 :: (and 0777777 $1) to 0000007
        |transform( 17.67): (and 0777777 0000007) => (and %mask (rot scount{7 8} tlatch{7 8}))
         andcommut( 11.00)
         applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 0000007)
          transform( 17.67): (and 0000007 0777777) => (and %mask (rot scount{7 8} tlatch{7 8}))
           andcommut( 11.00)cperandmatch( 11.00)
           decomposing by operand
            transform( 0.00): 0000007 => %mask
            |attempting constant match
            | it's a match!!
            |... success on transform( 0.00) with  0.00
            transform( 17.67): 0777777 => (rot scount{7 8} tlatch{7 8})
            |rotid( 9.00)
            |applying rotid: $1 :: (rot 0000000 $1) to 0777777
             transform( 17.67): (rot 0000000 0777777) => (rot scount{7 8} tlatch{7 8})
              operandmatch( 9.00)
              decomposing by operand
               transform( 2.84): 0000000 => scount{7 8}
                applying fetch decomposition
                |search( 2.84): (<- scount{7 8} 0000000)
                | shift( 2.00)
                | feasible: shift = (<- scount{4 9} %wild)
                |  transform( 0.00): 0000000 => %wild
                |    (using previous result)
                |    ... success on transform( 0.00) with  0.00
                | ... success on search( 2.84) with  2.00
                ... success on transform( 2.84) with  2.00
               transform( 14.83): 0777777 => tlatch{7 8}
                applying fetch decomposition
                |search( 14.83): (<- tlatch{7 8} 0777777)
                | ld.tl( 7.00)
                | feasible: ld.tl = (<- tlatch{6 9999} abus{5 6})
                |  transform( 13.83): 0777777 => abus{5 6}
                |   applying fetch decomposition
                |    search( 13.83): (<- abus{5 6} 0777777)
                |    |abus.gpr( 12.00)abus.fbus( 6.00)abus.dram( 8.00)
                |    |feasible: abus.fbus = (<- abus{5 12} fbus{2 3})
                |    | transform( 10.83): 0777777 => fbus{2 3}
                |    |  applying fetch decomposition
                |    |   search( 10.83): (<- fbus{2 3} 0777777)
                |    |    fbus.ones( 3.00)
                |    |    feasible: fbus.ones = (<- fbus{2 11} 0777777)
                |    |    ... success on search( 10.83) with  3.00
                |    | ... success on transform( 10.83) with  3.00
                |    |feasible: abus.dram = (<- abus{5 12} dram{4 5}[dadr{4 5} %wild])
                |    | transform( 8.62): 0777777 => dram{4 5}[dadr{4 5} %wild]
                |    |  applying fetch decomposition
                |    |   search( 8.62): (<- dram{4 5}[dadr{4 5} %wild] 0777777)
                |    |    ld.d.fbus( 5.00)
                |    |    feasible: ld.d.fbus = (<- dram{8 9999}[dadr{2 3} %wild] fbus{7 8})
                |    |    |transform( 0.00)*: dram{8 9999}[dadr{2 3} %wild] => dram{4 5}[dadr{4 5} %wild]
                |    |    | ... can't allocate resource!
                |    |    | ... fail on transform( 0.00)
                |    |    ... cutoff reached.
                |    |    ... fail on search( 8.62)
                |    |   ... fail on transform( 8.62)
                |    |... success on search( 13.83) with  6.00
                |   ... success on transform( 13.83) with  6.00
                | ... success on search( 14.83) with  7.00
                ... success on transform( 14.83) with  7.00
               ... success on transform( 17.67) with  9.00
            |... success on transform( 17.67) with  9.00
            applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0000007 0777777)
             transform( 14.84): (and 0777777 0000007) => (and %mask (rot scount{7 8} tlatch{7 8}))
             |... found previous failure
             |... fail on transform( 14.84)
            ... success on transform( 17.67) with  9.00
           ... success on transform( 17.67) with  9.00
          ... success on search( 19.67) with  11.00
         |... success on transform( 19.67) with  11.00
        applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0000007 0777777)
         transform( 18.65): (and 0777777 0000007) => (and areg breg)
         |... found previous failure
         |... fail on transform( 18.65)
        ... success on transform( 22.20) with  13.00
       ... success on transform( 22.20) with  13.00
      success on transform( 22.20) with  13.00
feasible: fbus.or = (<- fbus{2 11} (or areg breg))
 transform( 18.17): 0000007 => (or areg breg)
  orid( 16.00)
  applying orid: $1 :: (or 0000000 $1) to 0000007
   transform( 18.17): (or 0000000 0000007) => (or areg breg)
    andcommut( 16.00)
```

```
    | applying orcommut: (or $1 $2) :: (or $2 $1) to (or 0000000 0000007)
    |  transform( 18.17): (or 0000007 0000000) => (or areg breg)
    |   orcommut( 16.00)operandmatch( 16.00)
    |   decomposing by operand
    |    transform( 6.60): 0000000 => breg
    |    |applying fetch decomposition
    |    | search( 6.60): (<- breg 0000000)
    |    |  breg.fbl( 6.00)
    |    |  feasible: breg.fbl = (<- breg{4 13} fblatch{3 4})
    |    |   transform( 4.60): 0000000 => fblatch{3 4}
    |    |    applying fetch decomposition
    |    |    |search( 4.60): (<- fblatch{3 4} 0000000)
    |    |    | ld.fbl( 4.00)
    |    |    | feasible: ld.fbl = (<- fblatch{3 9999} fbus{2 3})
    |    |    |  transform( 3.60): 0000000 => fbus{2 3}
    |    |    |   applying fetch decomposition
    |    |    |    search( 3.60): (<- fbus{2 3} 0000000)
    |    |    |    |fbus.zero( 3.00)
    |    |    |    |feasible: fbus.zero = (<- fbus{2 11} 0000000)
    |    |    |    |... success on search( 3.60) with  3.00
    |    |    |   ... success on transform( 3.60) with  3.00
    |    |    | ... success on search( 4.60) with  4.00
    |    |   ... success on transform( 4.60) with  4.00
    |    |  ... success on search( 6.60) with  6.00
    |    |... success on transform( 6.60) with  6.00
    |    transform( 11.57): 0000007 => areg
    |    | (using previous result)
    |    |... success on transform( 11.57) with  11.00
    |   ... success on transform( 18.17) with  17.00
    |  ... success on transform( 18.17) with  17.00
    | ... success on transform( 18.17) with  17.00
    |... success on search( 25.20) with  18.00
  ... success on search( 45.60) with  25.00
51 nodes examined.
Maximum search depth: 19
Maximum axiom depth: 5
Approximate execution time: 1.89 seconds

Compacting set 0:
abus.linc  (0)
ld.tl  abus.fbus  fbus.ones  breg.ones  areg.mask 0000007 shift 0000000 ld.dmask 0000000 ld.dr.aset 0000000 (1)
fbus.and  (2)
  ... size 3, spread 18, cost 25

Compacting set 1:
ld.fbl  fbus.zero  abus.linc  (0)
ld.tl  abus.fbus  fbus.ones  breg.fbl  areg.mask 0000007 shift 0000000 ld.dmask 0000000 ld.dr.aset 0000000 (1)
fbus.or  (2)
  ... size 3, spread 20, cost 29

Modifying tables:
inner product is 7
fbus: 3, 1 => 8
tlatch: 1, 1 => 3
abus: 3, 1 => 8
carryout1: 0, 1 => 0
carryout2: 0, 1 => 0
carryout3: 0, 1 => 0

search( 57.60): (: (<- dram[dadr 0000000] lincwd) (<- fbus 0000007))
 decomposition( 48.00)
 search( 26.40): (<- dram[dadr 0000000] lincwd)
  ld.dr.aset( 22.00)ld.dr.aclr( 22.00)
  feasible: ld.dr.aset = (<- dram{3 9999}[dadr{2 3} %wild] (or dmask{1 2} abus))
   transform( 0.00)*: dram{3 9999}[dadr{2 3} %wild] => dram[dadr 0000000]
    transform( 0.00): 0000000 => %wild
    |attempting constant match
    | it's a match!!
    |... success on transform( 0.00) with  0.00
    ... success on transform( 0.00) with  0.00
   transform( 24.40): lincwd => (or dmask{1 2} abus)
    orid( 12.00)
    applying orid: $1 :: (or 0000000 $1) to lincwd
    |transform( 24.40): (or 0000000 lincwd) => (or dmask{1 2} abus)
    | operandmatch( 12.00)
    | decomposing by operand
    |  transform( 0.00): 0000000 => dmask{1 2}
    |   applying fetch decomposition
    |    search( 0.00): (<- dmask{1 2} 0000000)
    |    |ld.dmask( 0.00)
    |    |feasible: ld.dmask = (<- dmask{0 7} %bitset)
    |    | transform( 0.00): 0000000 => %bitset
    |    |  attempting constant match
    |    |   it's a match!!
    |    |   ... success on transform( 0.00) with  0.00
    |    |... success on search( 0.00) with  0.00
    |   ... success on transform( 0.00) with  0.00
    |  transform( 24.40): lincwd => abus
    |   applying fetch decomposition
    |    search( 24.40): (<- abus lincwd)
    |    |abus.linc( 12.00)
    |    |feasible: abus.linc = (<- abus{5 12} lincwd{4 5})
    |    |... success on search( 24.40) with  12.00
    |    ... success on transform( 24.40) with  12.00
    | ... success on transform( 24.40) with  12.00
    ... success on transform( 24.40) with  12.00
  feasible: ld.dr.aclr = (<- dram{3 9999}[dadr{2 3} %wild] (and (not dmask{1 2}) abus))
   transform( 0.00)*: dram{3 9999}[dadr{2 3} %wild] => dram[dadr 0000000]
    |(using previous result)
    ... success on transform( 0.00) with  0.00
   transform( 20.18): lincwd => (and (not dmask{1 2}) abus)
    No takers!
    ... cutoff reached.
    ... fail on transform( 20.18)
```

```
  ... success on search( 26.40) with  14.00
 search( 31.20): (<- fbus 0000007)
  fbus.and( 26.00)fbus.or( 26.00)fbus.xor( 26.00)
  feasible: fbus.and = (<- fbus{2 11} (and areg breg))
   transform( 23.20): 0000007 => (and areg breg)
    andid( 12.00)
    applying andid: $1 :: (and 0777777 $1) to 0000007
    |transform( 23.20): (and 0777777 0000007) => (and areg breg)
     andcommut( 12.00)
     applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 0000007)
      transform( 23.20): (and 0000007 0777777) => (and areg breg)
       andcommut( 12.00)operandmatch( 12.00)
       decomposing by operand
        transform( 2.58): 0777777 => breg
        |applying fetch decomposition
        | search( 2.58): (<- breg 0777777)
        |  breg.ones( 2.00)
        |  feasible: breg.ones = (<- breg{4 13} 0777777)
        |  ... success on search( 2.58) with  2.00
        |... success on transform( 2.58) with  2.00
       transform( 20.62): 0000007 => areg
       |applying fetch decomposition
       | search( 20.62): (<- areg 0000007)
       |  areg.mask( 10.00)
       |  feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
       |   transform( 18.62): 0000007 => (and %mask (rot scount{7 8} tlatch{7 8}))
       |    No takers!
       |    ... cutoff reached.
       |    ... fail on transform( 18.62)
       |  ... cutoff reached.
       |  ... fail on search( 20.62)
       |... fail on transform( 20.62)
      applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0000007 0777777)
       transform( 23.20): (and 0777777 0000007) => (and areg breg)
       |... found previous failure
       |... fail on transform( 23.20)
      ... cutoff reached.
      ... fail on transform( 23.20)
    ... cutoff reached.
    ... fail on transform( 23.20)
   ... cutoff reached.
   ... fail on transform( 23.20)
  feasible: fbus.or = (<- fbus{2 11} (or areg breg))
   transform( 23.20): 0000007 => (or areg breg)
    No takers!
    ... cutoff reached.
    ... fail on transform( 23.20)
  feasible: fbus.xor = (<- fbus{2 11} (xor areg breg))
   transform( 23.20): 0000007 => (xor areg breg)
    No takers!
    ... cutoff reached.
    ... fail on transform( 23.20)
  ... cutoff reached.
  ... fail on search( 31.20)
 ... cutoff reached.
 ... fail on search( 57.60)
25 nodes examined.
Maximum search depth: 8
Maximum axiom depth: 3
Approximate execution time: 1.53 seconds

search( 63.36): (: (<- dram[dadr 0000000] lincwd) (<- fbus 0000007))
 decomposition( 48.00)
 search( 29.04): (<- dram[dadr 0000000] lincwd)
  ld.dr.aset( 22.00)ld.dr.aclr( 22.00)
  feasible: ld.dr.aset = (<- dram{3 9999}[dadr{2 3} %wild] (or dmask{1 2} abus))
   transform( 0.00)*: dram{3 9999}[dadr{2 3} %wild] => dram[dadr 0000000]
    |(using previous result)
    ... success on transform( 0.00) with  0.00
   transform( 27.04): lincwd => (or dmask{1 2} abus)
    orid( 12.00)
    applying orid: $1 :: (or 0000000 $1) to lincwd
    |transform( 27.04): (or 0000000 lincwd) => (or dmask{1 2} abus)
    | orid( 24.00)operandmatch( 12.00)
    | decomposing by operand
    |  transform( 0.00): 0000000 => dmask{1 2}
    |    (using previous result)
    |    ... success on transform( 0.00) with  0.00
    |  transform( 27.04): lincwd => abus
    |   applying fetch decomposition
    |    search( 27.04): (<- abus lincwd)
    |    |abus.linc( 12.00)
    |    |feasible: abus.linc = (<- abus{5 12} lincwd{4 5})
    |    |... success on search( 27.04) with  12.00
    |    ... success on transform( 27.04) with  12.00
    | ... success on transform( 27.04) with  12.00
    ... success on transform( 27.04) with  12.00
  feasible: ld.dr.aclr = (<- dram{3 9999}[dadr{2 3} %wild] (and (not dmask{1 2}) abus))
   transform( 0.00)*: dram{3 9999}[dadr{2 3} %wild] => dram[dadr 0000000]
    |(using previous result)
    ... success on transform( 0.00) with  0.00
   transform( 22.39): lincwd => (and (not dmask{1 2}) abus)
    No takers!
    ... cutoff reached.
    ... fail on transform( 22.39)
  ... success on search( 29.04) with  14.00
 search( 34.32): (<- fbus 0000007)
  fbus.and( 26.00)fbus.or( 26.00)fbus.xor( 26.00)
  feasible: fbus.and = (<- fbus{2 11} (and areg breg))
   transform( 26.32): 0000007 => (and areg breg)
    andid( 12.00)
    applying andid: $1 :: (and 0777777 $1) to 0000007
    |transform( 26.32): (and 0777777 0000007) => (and areg breg)
    | andcommut( 12.00)operandmatch( 24.00)
```

```
| applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0777777 0000007)
 transform( 26.32): (and 0000007 0777777) => (and areg breg)
  andcommut( 12.00)operandmatch( 12.00)
  decomposing by operand
   transform( 2.74): 0777777 => breg
   |applying fetch decomposition
    search( 2.74): (<- breg 0777777)
     breg.ones( 2.00)
     feasible: breg.ones = (<- breg{4 13} 0777777)
      ... success on search( 2.74) with  2.00
    |... success on transform( 2.74) with  2.00
   transform( 23.58): 0000007 => areg
   |applying fetch decomposition
    search( 23.58): (<- areg 0000007)
     areg.mask( 10.00)
     feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
      transform( 21.58): 0000007 => (and %mask (rot scount{7 8} tlatch{7 8}))
       No takers!
       ... cutoff reached.
       ... fail on transform( 21.58)
      ... cutoff reached.
      ... fail on search( 23.58)
    |... fail on transform( 23.58)
  applying andcommut: (and $1 $2) :: (and $2 $1) to (and 0000007 0777777)
   transform( 26.32): (and 0777777 0000007) => (and areg breg)
   |... found previous failure
   |... fail on transform( 26.32)
  ... cutoff reached.
  ... fail on transform( 26.32)
 decomposing by operand
  transform( 10.80): 0777777 => areg
   applying fetch decomposition
   search( 10.80): (<- areg 0777777)
   |areg.mask( 10.00)
   |feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
    transform( 8.80): 0777777 => (and %mask (rot scount{7 8} tlatch{7 8}))
     No takers!
     ... cutoff reached.
     ... fail on transform( 8.80)
    |... cutoff reached.
    |... fail on search( 10.80)
  ... fail on transform( 10.80)
 ... cutoff reached.
| ... fail on transform( 26.32)
. . cutoff reached.
... fail on transform( 26.32)
feasible: fbus.or = (<- fbus{2 11} (or areg breg))
 transform( 26.32): 0000007 => (or areg breg)
  orid( 24.00)
  applying orid: $1 :: (or 0000000 $1) to 0000007
  |transform( 26.32): (or 0000000 0000007) => (or areg breg)
   orcommut( 21.00)operandmatch( 24.00)
   applying orcommut: (or $1 $2) :: (or $2 $1) to (or 0000000 0000007)
    transform( 26.32): (or 0000007 0000000) => (or areg breg)
     orcommut( 24.00)operandmatch( 21.00)
     decomposing by operand
      transform( 12.42): 0000007 => areg
      |... found previous failure
      |... fail on transform( 12.42)
     applying orcommut: (or $1 $2) :: (or $2 $1) to (or 0000007 0000000)
      transform( 26.32): (or 0000000 0000007) => (or areg breg)
      |... found previous failure
      |... fail on transform( 26.32)
     ... cutoff reached.
     ... fail on transform( 26.32)
   decomposing by operand
    transform( 10.80): 0000000 => areg
     applying fetch decomposition
     search( 10.80): (<- areg 0000000)
     |areg.mask( 10.00)
     |feasible: areg.mask = (<- areg{8 15} (and %mask (rot scount{7 8} tlatch{7 8})))
      transform( 8.80): 0000000 => (and %mask (rot scount{7 8} tlatch{7 8}))
       zeroand( 0.89)
       applying zeroand: 0000000 :: (and 0000000 ???) to 0000000
        transform( 8.80): (and 0000000 ???) => (and %mask (rot scount{7 8} tlatch{7 8}))
         operandmatch( 0.00)
         decomposing by operand
         |transform( 0.00): 0000000 => %mask
         | attempting constant match
         |  it's a match!!
         | ... success on transform( 0.00) with  0.00
         ... success on transform( 8.80) with  0.00
        ... success on transform( 8.80) with  0.00
       |... success on search( 10.80) with  2.00
      ... success on transform( 10.80) with  2.00
    transform( 15.52): 0000007 => breg
     applying fetch decomposition
     search( 15.52): (<- breg 0000007)
     |breg.con( 14.00)
     |feasible: breg.con = (<- breg{4 13} (@2 0000010 conhi{3 4} conlo{3 4}))
      transform( 13.52): 0000007 => (@2 0000010 conhi{3 4} conlo{3 4})
       con-unfold( 8.00)
       applying con-unfold to 0000007
        transform( 13.52): (@2 0000010 0000000 0000007) => (@2 0000010 conhi{3 4} conlo{3 4})
         operandmatch( 8.00)
         decomposing by operand
         |transform( 6.76): 0000000 => conhi{3 4}
         | applying fetch decomposition
          search( 6.76): (<- conhi{3 4} 0000000)
           ld.conhi( 4.00)
           feasible: ld.conhi = (<- conhi{0 9999} %wild)
            transform( 0.00): 0000000 => %wild
            | (using previous result)
            |... success on transform( 0.00) with  0.00
```

```
|  |  |  |  ... success on search( 6.76) with  4.00
|  |  |  ... success on transform( 6.76) with  4.00
|  |  transform( 6.76): 0000007 => conlo{3 4}
|  |   applying fetch decomposition
|  |    search( 6.76): (<- conlo{3 4} 0000007)
|  |     ld.conlo( 4.00)
|  |     feasible: ld.conlo * (<- conlo{0 9999} %wild)
|  |      transform( 0.00): 0000007 => %wild
|  |       attempting constant match
|  |       it's a match!!
|  |       ... success on transform( 0.00) with  0.00
|  |     ... success on search( 6.76) with  4.00
|  | ... success on transform( 6.76) with  4.00
|  ... success on transform( 13.52) with  8.00
| ... success on transform( 13.52) with  8.00
... success on search( 15.52) with  10.00
... success on transform( 15.52) with  10.00
... success on transform( 26.32) with  12.00
... success on transform( 26.32) with  12.00
feasible: fbus.xor * (<- fbus{2 11} (xor areg breg))
 transform( 24.20): 0000007 => (xor areg breg)
  xorid( 12.00)
  applying xorid: $1 :: (xor 0000000 $1) to 0000007
  transform( 24.20): (xor 0000000 0000007) => (xor areg breg)
   operandmatch( 12.00)
   decomposing by operand
    transform( 2.63): 0000000 => areg
     (using previous result)
     ... success on transform( 2.63) with  2.00
    transform( 21.57): 0000007 => breg
     applying fetch decomposition
      search( 21.57): (<- breg 0000007)
      breg.con( 14.00)
      feasible: breg.con * (<- breg{4 13} (@2 0000010 conhi{3 4} conlo{3 4}))
       transform( 19.57): 0000007 => (@2 0000010 conhi{3 4} conlo{3 4})
        con-unfold( 8.00)
        applying con-unfold to 0000007
         transform( 19.57): (@2 0000010 0000000 0000007) => (@2 0000010 conhi{3 4} conlo{3 4})
          operandmatch( 8.00)
          decomposing by operand
           transform( 9.78): 0000000 => conhi{3 4}
            applying fetch decomposition
             search( 9.78): (<- conhi{3 4} 0000000)
              ld.conhi( 4.00)
              feasible: ld.conhi * (<- conhi{0 9999} %wild)
               transform( 0.00): 0000000 => %wild
               (using previous result)
               ... success on transform( 0.00) with  0.00
              ... success on search( 9.78) with  4.00
            ... success on transform( 9.78) with  4.00
           transform( 9.78): 0000007 => conlo{3 4}
            applying fetch decomposition
             search( 9.78): (<- conlo{3 4} 0000007)
              ld.conlo( 4.00)
              feasible: ld.conlo * (<- conlo{0 9999} %wild)
               transform( 0.00): 0000007 => %wild
               (using previous result)
               ... success on transform( 0.00) with  0.00
              ... success on search( 9.78) with  4.00
            ... success on transform( 9.78) with  4.00
           ... success on transform( 19.57) with  8.00
          ... success on transform( 19.57) with  8.00
         ... success on search( 21.57) with  10.00
        ... success on transform( 21.57) with  10.00
       ... success on transform( 24.20) with  12.00
      ... success on search( 34.32) with  20.00
     ... success on search( 63.36) with  34.00
56 nodes examined.
Maximum search depth: 11
Maximum axiom depth: 3
Approximate execution time: 1.93 seconds

Compacting set 0:
abus.linc  (0)
ld.dmask 0000000 ld.dr.aset 0000000 (1)
ld.conhi 0000000 (2)
ld.conlo 0000007 areg.mask 0000000 breg.con  (3)
fbus.or  (4)
  ... size 5, spread 38, cost 24

Compacting set 1:
abus.linc  (0)
ld.dmask 0000000 ld.dr.aset 0000000 (1)
ld.conhi 0000000 (2)
ld.conlo 0000007 areg.mask 0000000 breg.con  (3)
fbus.xor  (4)
  ... size 5, spread 38, cost 24
```

References

[Aho 74] Aho, A. V., Hopcroft, J. E., and Ullman J. D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, Reading, Massachusetts, 1974.

[Aho 77] Aho, A. V., and Ullman, J. D.
Principles of Compiler Design.
Addison-Wesley, Reading, Massachusetts, 1977.

[Banerjee 79] Banerjee, U., Shen, S., Kuck, D. J., and Towle, R. A.
Time and Parallel Processor Bounds for Fortran-Like Loops.
IEEE Transactions on Computers C-28(9):660–670, September, 1979.

[Barbacci 77] Barbacci, M., Barnes, G., Cattell, R., and Siewiorek, D.
The ISPS Computer Description Language.
Technical Report, Carnegie-Mellon University, August, 1977.

[Bell 78] Bell, C.G., Kotok, A., Hastings, T.N., and Hill, R.
The Evolution of the DECSystem-10.
In Bell, C.G., Mudge, J.C., and McNamara, J.E. (editor), *Computer Engineering: A DEC View of Hardware Systems Design*, chapter 21.
Digital Press, Bedford, Massachusetts, 1978.

[Carter 78] Carter, W. C., Joyner, W. H., and Brand, D.
Microprogram Verification Considered Necessary.
In *Proc. National Computer Conference*, pages 657–664. AFIPS, June, 1978.

[Cattell 78] Cattell, R. G. G.
Formalization and Automatic Derivation of Code Generators.
PhD thesis, Carnegie-Mellon University, April, 1978.
Updated version published under the same title by UMI Research Press, Ann Arbor, 1982.

[Cocke 70] Cocke, J., and Schwartz, J. T.
Programming Languages and Their Compilers.
Technical Report, New York University, April, 1970.

[Dasgupta 76] Dasgupta, S., and Tartar, J.
The Identification of Maximal Parallelism in Straight-Line Microprograms.
IEEE Transactions on Computers C-25(10):986–992, October, 1976.

[Dasgupta 77] Dasgupta, S.
Parallelism in Loop Free Microprograms.
In Gilchrist, B. (editor), *Information Processing 77 (Proc. IFIP Congress 1977)*. North-Holland, Amsterdam, 1977.

[Dasgupta 78] Dasgupta, S.
Towards a Microprogramming Language Schema.
In *Proc. 11th Annual Workshop on Microprogramming*, pages 144–153. IEEE, November, 1978.

[Davidson 78] Davidson, S., and Shriver, B. D.
An Overview of Firmware Engineering.
Computer 11(5):21–33, May, 1978.

[Davidson 81] Davidson, S., Landskov, D., Shriver, B. D., and Mallett, P. W.
Some Experiments in Local Microcode Compaction for Horizontal Machines.
IEEE Transactions on Computers C-30(7):460–477, July, 1981.

[DeWitt 76] DeWitt, D. J.
A Machine Independent Approach to the Production of Optimized Horizontal Microcode.
PhD thesis, University of Michigan, June, 1976.

[Digital 78] Digital Equipment Corporation.
Microcomputer Processors.
Digital Equipment Corporation, Maynard, Massachusetts, 1978.

[Erman 78] Erman, L. D., and Lesser, V. R.
The Hearsay-II System: A Tutorial.
In Lea, W. A. (editor), *Trends in Speech Recognition*, chapter 16. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Fagg 64] Fagg, P., Brown, J. L., Hipp, J. A., and Doody, D. T.
IBM System/360 Engineering.
In *Proc. Spring Joint Computer Conference*, pages 205–231. AFIPS, 1964.

[Fisher 79] Fisher, J. A.
The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources.
PhD thesis, New York University, October, 1979.

[Fisher 80] Fisher, J. A.
2^n-Way Jump Microinstruction Hardware and an Effective Instruction Binding Method.
In *Proc. 13th Annual Workshop on Microprogramming*, pages 64–75. IEEE, November, 1980.

[Fisher 81a] Fisher, J. A.
Trace-Scheduling: A Technique for Global Microcode Compaction.
IEEE Transactions on Computers C-30(7):478–490, July, 1981.

[Fisher 81b] Fisher, J. A., Landskov, D., and Shriver, B. D.
Microcode Compaction: Looking Backward and Forward.
In *Proc. National Computer Conference*, pages 95–102. AFIPS, 1981.

[FPS 82] FPS Technical Publications Staff.
APAL64 Programmer's Reference Manual.
Floating Point Systems, 1982.

[Fuller 76] Fuller, S. H., Almes, G. T., Broadley, W. H., Forgy, C. L., Karlton, P. L., Lesser, V. R., and Teter, J. R.
PDP-11/40E Microprogramming Reference Manual.
Technical Report, Carnegie-Mellon University, January, 1976.

[Garey 79] Garey, M. R., and Johnson, D. S.
Computers and Intractability: A Guide to the Theory of NP-Completeness.
Freeman, San Francisco, 1979.

[Gosling 81] Gosling, J.
Some Issues and Techniques for Microcode Compilers (unpublished report).
1981.

[Grishman 78] Grishman, R.
The Structure of the Puma Computer System: Overview and the Central Processor.
Technical Report C00-3077-157, New York University, November, 1978.

[Hansen 80] Hansen, I., and Leszczylowski, J.
On Fundamentals of Computer-Aided Design of Firmware.
In *Proc. 13th Annual Workshop on Microprogramming*, pages 3–12. IEEE, November, 1980.

[Holloway 79] Holloway, J., Steele, G. L. Jr., Sussman, G. J. and Bell, A.
The SCHEME-79 Chip.
AI Memo 559, MIT Artificial Intelligence Laboratory, December, 1979.

[Horowitz 78] Horowitz, E., and Sahni, S.
Fundamentals of Computer Algorithms.
Computer Science Press, Potomac, Maryland, 1978.

[Husson 70] Husson, S. S.
Microprogramming: Principles and Practice.
Prentice-Hall, Englewood Cliffs, New Jersey, 1970.

[Johannsen 78] Johannsen, D.
Our Machine, A Microcoded LSI Processor.
In *Proc. 11th Annual Workshop on Microprogramming*, pages 1–7. IEEE, November, 1978.

[Jones 79] Jones, A. K., Chansler, R. J., Durham, I., Schwans, K., and Vegdahl, S. R.
STAROS, A Multiprocessor Operating System for the Support of Task Forces.
In *Proc. 7th Symposium on Operating Systems Principles*, pages 117–127. ACM/SIGOPS, December, 1979.

[Jones 80] Jones, A. K., and Gehringer, E. F., editors.
The Cm Multiprocessor Project: A Research Review.*
Technical Report CMU-CS-80-131, Carnegie-Mellon University, July, 1980.

[Kernighan 78] Kernighan, B. W., and Ritchie, D. M.
The C Programming Language.
Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Kim 79] Kim, J., and Tan, C. J.
Register Assignment Algorithms for Optimizing Micro-Code Compilers—Part 1.
Technical Report RC 7639, IBM Thomas J. Watson Research Center, May, 1979.

[Landskov 80] Landskov, D., Davidson, S., Shriver, B., and Mallett, P. W.
Local Microcode Compaction Techniques.
ACM Computing Surveys 12(3):261–294, September, 1980.

[Leverett 79] Leverett, B. W., Cattell, R. G. G., Hobbs, S. O., Newcomer, J. M., Reiner, A. H., Schatz, B. R., and Wulf, W. A.
An Overview of the Production Quality Compiler-Compiler Project.
Technical Report CMU-CS-79-105, Carnegie-Mellon University, February, 1979.

[Leverett 81] Leverett, B. W.
Register Allocation in Optimizing Compilers.
PhD thesis, Carnegie-Mellon University, February, 1981.

[Lowry 69] Lowry, E., and Medlock, C. W.
Object Code Optimization.
Communications of the ACM 12(1):13–22, January, 1969.

[Ma 80] Ma, P. Y., and Lewis, T. G.
Design of a Machine-Independent Optimizing System for Emulator Development.
ACM Transactions on Programming Languages and Systems 2(2):239–262, April, 1980.
A revision was published under title "On The Design of a Microcode Compiler for a Machine-Independent High-Level Language" in the May 1981 issue of IEEE Transactions of Software Engineering.

[Mallett 78] Mallett, P. W.
Methods of Compacting Microprograms.
PhD thesis, University of Southwestern Louisiana, December, 1978.

[Marwedel 81] Marwedel, P.
A Retargetable Microcode Generation System for a High-Level Microprogramming Language.
In *Proc. 14th Annual Workshop on Microprogramming*, pages 115–123. IEEE, December, 1981.

[McCreight 80] McCreight, E. M.
Personal Communication.
1980.

[Meyers 80] Meyers, W. J.
Design of a Microcode Link Editor.
In *Proc. 13th Annual Workshop on Microprogramming*, pages 165–170. IEEE, November, 1980.

[Michie 68] Michie, D.
"Memo" Functions and Machine Learning.
Nature 218:19–22, April, 1968.

[Mueller 80a] Mueller, R. A.
Automated Microprogram Synthesis.
PhD thesis, Univerisity of Colorado, 1980.

[Mueller 80b] Mueller, R. A.
Formalization and Automated Synthesis of Microprograms.
In *Proc. 13th Annual Workshop on Microprogramming*, pages 45–53. IEEE, November, 1980.

[Nanodata 72] Nanodata Computer Corp.
QM-1 Hardware-Level Users Manual.
Nanodata Computer Corp., Buffalo, New York, 1972.

[Nilsson 80] Nilsson, N. J.
Principles of Artificial Intelligence.
Tioga, Palo Alto, California, 1980.

[Ousterhout 78] Ousterhout, J. K.
Cm Kmap Microprogramming Manual and Debugger Manual.*
Carnegie-Mellon University, 1978.

[Parker 81] Parker, A. C., and Wilner, W. T.
Microprogramming—The Challenges of VLSI.
In *Proc. National Computer Conference*, pages 63–68. AFIPS, 1981.

[Patterson 76] Patterson, D. A.
STRUM: Structured Microprogramming System for Correct Firmware.
IEEE Transactions on Computers C-25(10):974–985, October, 1976.

[Patterson 79] Patterson, D. A., Lew, K., and Tuck R.
Towards an Efficient, Machine-Independent Language for Microprogramming.
In *Proc. 12th Annual Workshop on Microprogramming*, pages 22–35. IEEE, November, 1979.

[Poe 80] Poe, M. D.
Heuristics for the Global Optimization of Microprograms.
In *Proc. 13th Annual Workshop on Microprogramming*, pages 13–22. IEEE, December, 1980.

[Poe 81] Poe, M. D., Goodell, R., and Steely, S. Jr.
Issues of the Design of a Low Level Microprogramming Language for Global Microcode Compaction.
In *Proc. 14th Annual Workshop on Microprogramming*, pages 88–94. IEEE, October, 1981.

[Ramamoorthy 74] Ramamoorthy, C. V., and Tsuchiya, M.
A High-Level Language for Horizontal Microprogramming.
IEEE Transactions on Computers C-23(8):791–801, August, 1974.

[Rosen 79] Rosen, B.
PERQ Microprogrammers Guide.
Three Rivers Computer Corporation, 1979.

[Salisbury 76] Salisbury, A. B.
Microprogrammable Computer Architectures.
American Elsevier, New York, 1976.

[Sint 81] Sint, M.
MIDL—A Microinstruction Description Language.
In *Proc. 14th Annual Workshop on Microprogramming*, pages 95–106. IEEE, October, 1981.

[Slate 77] Slate, D.J., and Atkin, L.R.
Chess 4.5—The Northwestern University Chess Program.
In Frey, P.W. (editor), *Chess Skill in Man and Machine*, chapter 4. Springer-Verlag, New York, 1977.

[Strecker 78] Strecker, W.D.
Vax-11/780—A Virtual Addressing Extension of the DEC PDP-11 Family.
In *Proc. National Computer Conference*, pages 967–980. AFIPS, 1978.
Also published in "Computer Structures: Principles and Examples" by Siewiorek, Bell, and Newell; McGraw-Hill, 1982.

[Syiek 80] Syiek, D. A.
Personal Communication.
1980.

[Tan 78] Tan, C. J.
Code Optimization Techniques for Micro-Code Compilers.
Technical Report RC 6936, IBM Thomas J. Watson Research Center, January, 1978.

[Tokoro 78] Tokoro. M., Takizuka, E., Tamura, E and Yamaura, I.
A Technique of Global Optimizat[illegible]f Microprograms.
In *Proc. 11th Annual Workshop on Microprogramming*, pages 41–50. IEEE, 1978.

[Tokoro 81] Tokoro, M., Tamura, E., and Takizuka, T.
Optimization of Microprograms.
IEEE Transactions on Computers C-30(7):491–504, July, 1981.

[Tsuchiya 74] Tsuchiya, M., and Gonzalez, M. J.
An Approach to Optimization of Horizontal Microprograms.
In *Proc. 7th Annual Workshop on Microprogramming*, pages 85–90. IEEE, October, 1974.
Also published in IEEE Transactions on Computers (Oct. 1976) under the title "Toward Optimization of Horizontal Microprograms".

[Ulrich 80] Ulrich, J. W.
The Derivation of Microcode by Symbolic Execution.
In *Proc. 13th Annual Workshop on Microprogramming*, pages 38–42. IEEE, November, 1980.

[Vegdahl 81] Vegdahl, S. R., and Jones, A. K.
STAROS Microcode Wizard's Manual.
Technical Report, Carnegie-Mellon University, 1981.

[Wilkes 51] Wilkes, M. V.
The Best Way to Design an Automatic Calculating Machine.
In *Manchester University Computer Inaugural Conference.* Ferrante, London, July, 1951.

[Winston 77] Winston, P. H.
Artificial Intelligence.
Addison-Wesley, Reading, Massachusetts, 1977.

[Wood 79a] Wood, W. G.
The Computer Aided Design of Microprograms.
PhD thesis, University of Edinburgh, November, 1979.

[Wood 79b] Wood, W. G.
Global Optimization of Microprograms through Modular Control Constructs.
In *Proc. 12th Annual Workshop on Microprogramming*, pages 1–6. IEEE, November, 1979.

[Wulf 75] Wulf, W. A., Johnsson, R. K., Weinstock, C. B., Hobbs, S. O. and Geschke, C. M.
The Design of an Optimizing Compiler.
American Elsevier, New York, 1975.

[Yau 74] Yau, S. S., Schowe, A. C., and Tsuchiya, M.
On Storage Optimization of Horizontal Microprograms.
In *Proc. 7th Annual Workshop on Microprogramming*, pages 98–106. IEEE, October, 1974.

Index